Universidad
Carlos III de Madrid

Departamento de Ingeniería Telemática

TRABAJO FIN DE GRADO

Estudio de prestaciones del protocolo HTTP versión 2

Autor: Daniel Vega Jiménez

Tutor: Carlos García Rubio

Co-Tutor: Mª Celeste Campo Vázquez

Leganés, 20 de junio de 2016

Agradecimientos

En primer lugar me gustaría agradecer a Carlos y Celeste la oportunidad que me dieron de empezar una beca de investigación con ellos cuando todavía era alumno de Tercero. Agradecerles también toda la ayuda que me han prestado sin la cual habría estado mucho más perdido en la realización de este Trabajo de Fin de Grado.

Por otro lado agradecer a mis compañeros ya que sin ellos la carrera no habría sido la misma, en especial a Cristian, Jorge y Raúl por todas esas tardes que hemos pasado en la universidad haciendo trabajo, estudiando o simplemente pasando un buen rato.

Por último y no menos importante, me gustaría agradecer a mi familia todo el apoyo que me han dado estos cuatro años en los que han habido momentos difíciles y siempre han estado ahí para ayudarme. En especial a mis padres, mi hermana y mi novia que han tenido muchísima paciencia en las épocas de exámenes.

Índice general

1. Introduction and objectives **1**
1.1. Introduction 1
1.2. Objectives 2
1.3. Material resources 3
1.4. Structure of the report 3

2. Introducción y objetivos **5**
2.1. Introducción 5
2.2. Objetivos 6
2.3. Medios empleados 7
2.4. Estructura de la memoria 8

3. Marco regulador **9**

4. Estado del arte **11**
4.1. HTTP/2 11
4.1.1. Introducción 11
4.1.2. Características básicas del protocolo 13
4.1.2.1. Inicio HTTP/2 13
4.1.2.2. Frames HTTP/2 15
4.1.2.3. Principales tipos de Frames HTTP/2 15
4.1.2.4. Otros frames HTTP/2 23
4.1.2.5. Tipos de errores 25
4.1.3. Nuevas características del protocolo 26
4.1.3.1. Control de flujo en HTTP/2 26
4.1.3.2. Priorización de flujos 28
4.1.3.3. Server push 29
4.1.3.4. Protocolo binario 30
4.1.4. Otras características del protocolo 30
4.1.4.1. Intercambio de mensajes HTTP 30
4.1.4.2. Mecanismos de fiabilidad en HTTP/2 32

4.1.4.3. TLS . . . 33
4.1.4.4. Extensión del protocolo . . . 33
4.2. HPACK . . . 34
4.2.1. Introducción . . . 34
4.2.2. Proceso de codificación . . . 35
4.2.3. Representación de los campos de la cabecera . . . 36
4.2.4. Decodificación del bloque de cabeceras . . . 36
4.2.5. Gestión de la tabla dinámica . . . 36
4.2.6. Representaciones de tipo primitivo . . . 37
4.2.6.1. Números enteros . . . 37
4.2.6.2. Cadena de caracteres . . . 38
4.2.7. Seguridad del protocolo . . . 38
4.3. Estudios similares . . . 39
4.3.1. Estudios similares anteriores basados en SPDY . . . 39
4.3.2. Estudios similares en la actualidad basados en HTTP/2 40

5. Preparación del entorno de pruebas 41
5.1. Análisis de los clientes HTTP/2 . . . 41
5.2. Captura de paquetes HTTP/2 . . . 42
5.2.1. Capturas desde el navegador Web . . . 42
5.2.2. Capturas utilizando Wireshark . . . 45
5.2.2.1. Descifrado de paquetes en wireshark . . . 45
5.3. Análisis de los servidores HTTP/2 . . . 47
5.4. Puesta a punto del servidor Open Lite Speed . . . 48
5.4.1. Preparación de Wireshark para la captura de paquetes del servidor . . . 51

6. Metodología seguida para la realización de las pruebas 53
6.1. Creación de los archivos html para las pruebas . . . 54
6.2. Simulación del estado de la red . . . 55
6.2.1. Configuración del cliente . . . 55
6.2.2. Configuración del servidor . . . 56
6.2.3. Configuración del nodo intermedio . . . 56
6.3. Elaboración del script automático de pruebas . . . 58

7. Resultados 61
7.1. Retrasos en la red . . . 61
7.1.1. Resultados esperados . . . 61
7.1.2. Resultados obtenidos . . . 62
7.1.2.1. Imagen de 48KB . . . 63
7.1.2.2. Imagen de 578KB . . . 64

7.1.2.3. Imagen de 1.21MB 65
7.2. Pérdida de paquetes en la red 66
7.2.1. Resultados esperados 66
7.2.2. Resultados obtenidos 66
7.2.2.1. Imagen de 48KB 67
7.2.2.2. Imagen de 578KB 68
7.2.2.3. Imagen de 1.21MB 69
7.3. Variaciones de ancho de banda en la red 70
7.3.1. Resultados esperados 70
7.3.2. Resultados obtenidos 70
7.3.2.1. Imagen de 48KB 71
7.3.2.2. Imagen de 578KB 72
7.3.2.3. Imagen de 1.21MB 73
7.4. Eficiencia de HPACK 74
7.4.1. Resultados esperados 74
7.4.2. Resultados obtenidos 74
7.4.2.1. Imagen de 48 KB 75
7.4.2.2. Imagen de 578 KB 76
7.4.2.3. Imagen de 1.21 MB 77

8. Conclusions **79**
8.1. Achievement of objectives 79
8.2. Final conclusions 79
8.3. Future works 80

9. Conclusiones **81**
9.1. Consecución de objetivos 81
9.2. Conclusiones finales 81
9.3. Trabajos futuros 82

Referencias **83**

A. Planificación y presupuesto **85**
A.1. Planificación 85
A.1.1. Descomposición de tareas 85
A.1.2. Diagrama de Gantt 88
A.2. Presupuesto 90

Índice de figuras

4.1. ALPN-Inicio de negociación del cliente 14
4.2. ALPN-Contestación del servidor 14
4.3. Intercambio de paquetes en HTTP2 16
4.4. Frame SETTINGS 18
4.5. Frame WINDOW_UPDATE conexión 19
4.6. Frame WINDOW_UPDATE-2 flujo 19
4.7. Frame HEADERS 21
4.8. Frame DATA 22
4.9. Cabeceras Frame HEADERS 32

5.1. Captura de paquetes con Google Chrome-1 43
5.2. Captura de paquetes con Google Chrome-2 43
5.3. Captura de paquetes con Google Chrome-3 44
5.4. Configuración de wireshark-1 45
5.5. Configuración de wireshark-2 46
5.6. Ejemplo de captura 47
5.7. Configuración del servidor-1 49
5.8. Configuración del servidor-2 49
5.9. Configuración del servidor-3 50
5.10. Configuración del servidor-4 50
5.11. Configuración de la captura de paquetes del servidor 51

6.1. Entorno de red 55

7.1. Retraso con imagen de 48KB 63
7.2. Retraso con imagen de 578KB 64
7.3. Retraso con imagen de 1.21MB 65
7.4. Pérdidas con imagen de 48KB 67
7.5. Pérdidas con imagen de 578KB 68
7.6. Pérdidas con imagen de 1.21MB 69
7.7. Ancho de banda con imagen de 48KB 71
7.8. Ancho de banda con imagen de 578KB 72

7.9. Ancho de banda con imagen de 1.21MB 73
7.10. HPACK Imagen de 48KB 75
7.11. HPACK Imagen de 578KB 76
7.12. HPACK Imagen de 1.21MB 77

A.1. Tareas del diagrama de Gantt 88
A.2. Diagrama de Gantt 89

Índice de tablas

5.1. Navegadores compatibles con HTTP/2 42
5.2. Servidores compatibles con HTTP/2 48

6.1. Configuraciones de red 57

7.1. Tamaño de paquetes 48KB 75
7.2. Tamaño de paquetes 578KB 76
7.3. Tamaño de paquetes 1.21MB 77

Capítulo 1

Introduction and objectives

1.1. Introduction

HTTP/2 protocol stems from the need to improve HTTP version 1.1. One of the reasons why HTTP/1.1 needs to be reviewed is because actual webpages aren't as simple as they were when it was designed. Currently, most of webpages that we can find on the Internet are made up of a huge variety of resources. In addition, those resources are usually dynamic in order to personalize the webpage to each user so it adds a new level of complexity.

Nowadays webpages are more complex because of the large number of resources (images, javascript, css, etc) that they use. The average size of a webpage is about 2280 KB whereas only two years ago it was 1700 KB [1].

Furthermore, in recent years, the number of secure connections have risen. Looking back two years, only 8 % of the Internet connections were encrypted (HTTPS). Nowadays, 26 % of Internet connections are encrypted [1].

All this information allows us to guess that the protocol has the problem that it wasn't designed to work under encrypted connections neither to work with such a large number of resources. For example, surfing the Internet in an airport or in a restaurant with a mobile phone wasn't thought to be possible a few years ago. Another example, nowadays we can buy online tickets or use our bank account online. Due to these new possibilities, Internet connections tend to be encrypted so as to increase the security. Therefore, it is necessary to improve the efficiency of the HTTP protocol

in these new situations because Internet users have become accustomed to browse the internet at high speed and the current HTTPS protocol is safe but not very fast.

This made the researchers to think if it would be possible to improve the efficiency of the protocol when it is working under encrypted connections. Apparently, this was difficult because the protocol spent a lot of time negotiating, encrypting and decrypting the messages. However, Google thought it could be possible and, because of that, they started the development of a new protocol called SPDY. A short time later SPDY was renamed to HTTP/2 which intends to improve the loading time of Internet webpages and also fix the known security problems of HTTPS protocol.

This Final Bachelor Project pretends to analyse the performance of the new protocol in order to prove and quantify the improvements over the earlier protocol.

1.2. Objectives

The main objective of this Final Bachelor Project is to evaluate the efficiency of the HTTP/2 protocol compared to the earlier version of HTTP.

To reach this objective I have established the following objectives:

- Study of the HTTP/2 protocol.
- Study of the HPACK protocol.
- Analysis of web browsers that implements HTTP/2 and HPACK.
- Analysis of servers that implements HTTP/2 and HPACK.
- Deployment of a web server that implements both HTTP/2 and HTTPS.
- Deployment of a HTTP/2 client.
- Configuration of an environment that allows to capture the packets exchanged between the client and the server despite the fact that those packets are encrypted.
- Development of a script that makes the performance tests automatically.

- Evaluation of the performance of HTTP/2.
- Evaluation of the performance of HPACK.
- Preparation of the report of this Final Bachelor Project.

1.3. Material resources

The material resources used in this Final Bachelor Project have been:

- Laboratory in the Telematic department of the University UC3M. It was used as workstation.
- Computer Ubuntu 14.04.2 LTS. This computer was used to do the performance tests.
- Computer Windows 7. This computer was used to prepare the figures, documents and packet captures with Wireshark.
- Google Chrome 42.0.2311.135. Web browser used to capture packets.
- Software Chrome-HAR-Capturer. Library used to develop the script. This library get the time that Google Chrome spend loading a webpage.
- Ubuntu 12.04.5 LTS server. Server machine in which the HTTP/2 server was deployed.
- Servidor Web Open LiteSpeed WebServer 1.3.9.
- Virtual machine Ubuntu 12.04.5 LTS. Is the intermediate node.
- Software TC. This software is used to simulate network conditions.
- Software Netem. This software simulate network conditions.
- Whireshark 1.99.8 Beta. Software used to capture packets.

1.4. Structure of the report

1. **Introduction and objectives:** Chapter in which this Final Bachelor Project is introduced. The project is placed in the actual social context and the objectives are defined.
2. **State of art:** Describe the current situation of the studied protocols.

3. **Preparation of the testing environment:** Necessary previous configuration in order to do the tests.

4. **Methodology:** There is a detailed description about how the performance test is going to be executed. In addition, in this chapter is explained how the script works.

5. **Results:** In this chapter is located the main analysis of the protocols. There is a comparison between the theoretical and the real behaviour.

6. **Conclusions:** There is a final reflexion about the efficiency of both protocols. Moreover, I have proposed new working lines in order to increase my knowledge of HTTP/2 protocol.

7. **Work planning and budget:** In this chapter there is a task breakdown. Furthermore, there is a Gantt char in which task dependency is shown. At the end of this chapter is located the budget of the project.

Capítulo 2

Introducción y objetivos

2.1. Introducción

El protocolo HTTP/2 surge a partir de la idea de mejorar HTTP versión 1.1. Una de las razones por las que se pretende revisar es porque las páginas de hoy en día ya no son tan básicas como las que había cuando se diseñó HTTP/1.1. Actualmente, casi todas las webs presentes en Internet suelen estar compuestas por varios recursos. Los cuales suelen ser dinámicos de manera que la página se personaliza para cada usuario añadiendo un nuevo punto de complejidad.

Hoy en día, las páginas web utilizan un gran número de recursos (imágenes, javascript, css, etc). Debido a ello, en media, ya superan los 2280 KB de tamaño mientras que hace tan sólo 2 años estaba en los 1700 KB [1].

Por otro lado, en los últimos años se ha observado un aumento considerable en la utilización de conexiones seguras. Si echamos la vista dos años atrás tan sólo el 8 % de las conexiones eran cifradas (HTTPS) mientras que en la actualidad el 26 % ya lo son [1].

Toda esta información nos permite adivinar que un problema que tiene el protocolo actual es que no estaba diseñado para tales casos de sobrecarga a la hora de mostrar una página web, ni se ideó con la idea de que las conexiones fueran cifradas. Por ejemplo, hace unos años quizá no se planteó la posibilidad de que aeropuertos, cafeterías o restaurantes dieran la posibilidad a la gente de conectarse a Internet o que para comprar una entrada o manejar tu cuenta bancaria se pudiera hacer *online*. Debido a estas nuevas oportunidades, las conexiones a Internet tienden a encriptarse

para dar más seguridad al usuario final. Por lo tanto, es necesario mejorar la eficiencia del protocolo HTTP en estas nuevas situaciones puesto que los usuarios se han acostumbrado a una alta velocidad de carga en las páginas web y el protocolo HTTPS aunque es seguro, no es demasiado rápido.

Esto hizo que los investigadores se plantearan si sería posible mejorar la eficiencia del protocolo en los casos en los que las páginas web fueran cifradas dado que tanto el proceso de encriptación, como el de desencriptación requerían bastante tiempo. Google así lo creía y por ello comenzó el desarrollo de un protocolo llamado SPDY el cual se conoce actualmente como HTTP/2 que pretende reducir el tiempo de carga de las páginas de Internet y, a su vez, mejorar los problemas de seguridad conocidos para el protocolo HTTPS.

Este Trabajo de Fin de Grado, pretende analizar el funcionamiento de este nuevo protocolo para comprobar y cuantificar las mejoras de rendimiento que podría proporcionar frente a su versión anterior.

2.2. Objetivos

El objetivo principal de este trabajo es comprobar la eficiencia de HTTP/2 respecto a su antecesor HTTPS (HTTP/1.1 sobre TLS).

Para lograr el cumplimiento de ese objetivo principal, se establecen los siguientes objetivos previos a cumplir:

- Comprensión del protocolo HTTP/2.
- Compresión del protocolo HPACK.
- Estudio de los clientes que implementan HTTP/2.
- Estudio de los servidores que implementan HTTP/2.
- Despliegue de un servidor web con HTTP/2 y HTTPS contra el que realizar las pruebas.
- Despliegue de un cliente HTTP/2.
- Configuración de un entorno en el cliente que permita capturar los paquetes intercambiados aunque estos estén cifrados.
- Diseño de un script que realice las pruebas de rendimiento de forma automática.

- Análisis del rendimiento de HTTP/2.
- Análisis del rendimiento de HPACK.
- Elaboración de la memoria del Trabajo de Fin de Grado.

2.3. Medios empleados

Los medios empleados para este proyecto han sido:

- Laboratorio en el departamento de Telemática de la UC3M. Se utilizó como puesto de trabajo.
- Ordenador Ubuntu 14.04.2 LTS. Este ordenador sirvió para realizar todas las pruebas de rendimiento.
- Ordenador Windows 7. Este ordenador se utilizó para la elaboración de las gráficas, documentación y captura de paquetes con Wireshark.
- Google Chrome 42.0.2311.135. Navegador web utilizado para la captura de paquetes.
- Software Chrome-HAR-Capturer. Librería utilizada para desarrollar el script. Esta librería obtiene el tiempo que emplea Google Chrome en cargar una página.
- Servidor Ubuntu 12.04.5 LTS (Halffter). Máquina en la que se desplegó el servidor web.
- Servidor Web Open LiteSpeed WebServer 1.3.9.
- Máquina virtual Ubuntu 12.04.5 LTS. Es el nodo intermedio.
- Software TC. Sirve para simular condiciones en la red.
- Software Netem. Sirve para simular condiciones en la red.
- Whireshark 1.99.8 Beta. Software utilizado para la captura de paquetes.

2.4. Estructura de la memoria

1. **Introducción y objetivos:** Capítulo en el que se introduce brevemente la investigación en el contexto social actual y se establecen los objetivos a cumplir con esta investigación.

2. **Estado del arte:** Describe la situación actual de los protocolos a analizar.

3. **Preparación del entorno de pruebas:** Se especifica la configuración previa a la realización de las pruebas.

4. **Metodología seguida para la realización de las pruebas:** En este capítulo se detalla la estructura de las pruebas y cómo se van a realizar. Además, se explica la elaboración de un script que permite automatizar esas pruebas.

5. **Resultados:** Se realiza un análisis de los resultados obtenidos comparándolos con los resultados que se deberían obtener según un análisis previo teórico del protocolo.

6. **Conclusiones:** Se realiza una reflexión sobre la eficacia de ambos protocolos y sobre las nuevas lineas de investigación a seguir para profundizar en este análisis de rendimiento de HTTP/2.

7. **Planificación y presupuesto:** Se desglosan las tareas realizadas así como un diagrama de Gantt en el que se muestra la dependencias entre tareas. También se adjunta el presupuesto de este trabajo.

Capítulo 3

Marco regulador

Antes del comienzo de este trabajo se realizó un estudio del marco regulador vigente para comprobar que no había ningún impedimento legal a la hora de afrontar este proyecto.

Tras el análisis de la regulación actual, se han encontrado dos puntos que podrían afectar a la elaboración del proyecto:

- Ley 32/2003, de 3 de noviembre, General de Telecomunicaciones [3]. Dentro de la Ley general de telecomunicaciones el artículo 36 hace referencia a la posibilidad de transmitir por internet datos cifrados. Esto atañe a este trabajo puesto que todo el estudio se realiza sobre dos protocolos que hacen uso de transmisión de datos cifrados.
- IETF (Internet Engineering Task Force) [2]. Los protocolos utilizados en este trabajo, han sido definidos por este organismo el cual permite la utilización libre de los documentos que publican a modo de RFC. Mediante la publicación de las RFCs pretenden crear estandarizaciones que cualquiera puede seguir o ignorar voluntariamente.

Capítulo 4

Estado del arte

4.1. HTTP/2

4.1.1. Introducción

El protocolo web HTTP/2 [15] ha sido diseñado con el objetivo de sustituir y mejorar al protocolo web HTTP/1.1. Al comienzo de este trabajo (Febrero de 2015) el protocolo se encontraba en el draft-14. Actualmente, el protocolo ya ha sido publicado como una RFC (*Request For Comments*), más concretamente la RFC7540, la cual fue publicada el 14 de Mayo de 2015.

El protocolo HTTP/2 surge a raíz del protocolo web SPDY [13] [14] ideado por Google. Tras la publicación de SPDY, el grupo encargado de trabajo de HTTP del IETF (*Internet Engineering Task Force*) se dio cuenta de que el protocolo de Google estaba siendo utilizado por grandes proyectos como Mozilla o Nginx y que, además, tenía algunas mejoras de rendimiento respecto a HTTP/1.1. Por esto, se decidió que HTTP/2 debía basarse en SPDY.

Este protocolo está desarrollado por el grupo de trabajo de HTTP del IETF junto con los ingenieros de Firefox, Google, Twitter, Akamai, Curl, una sección de Microsoft y otros muchos desarrolladores que trabajaron en conjunto para crear implementaciones del protocolo en lenguajes como Python, Ruby, NodeJS, C++, etc.

El objetivo de HTTP/2 es mejorar algunas características respecto a su predecesor HTTP/1.1 que le ayuden a reducir tanto la carga de datos que viajan por la red, como la velocidad de carga de las páginas web manteniendo la compatibilidad con HTTP/1.1.

HTTP/1.1 permite realizar solicitudes utilizando *pipelining*. El *pipelining* consiste en la capacidad de transmitir varias solicitudes HTTP sin necesidad de esperar una respuesta para cada una de esas solicitudes. No obstante, esta característica aparece deshabilitada por defecto en la mayor parte de los navegadores web debido a que esta funcionalidad, lejos de mejorar el rendimiento, provoca que se produzcan errores en los navegadores, proxys y servidores web, por lo que recomiendan no habilitar esta funcionalidad. Al no poder utilizar *pipelining* se produce un error conocido como *Head of Line Blocking* que consiste en el bloqueo de varias peticiones al esperar a que la petición anterior reciba su respuesta ya que esas respuestas en HTTP/1.1 deben de recibirse ordenadas. En este caso, al tener que recibir una respuesta por petición, si alguna respuesta tarda mucho en llegar, no llega, o llega en distinto orden se bloquean todas las peticiones siguientes. Debido a ello, los protocolos HTTP anteriores a HTTP/2 necesitan hacer uso de múltiples conexiones TCP con los servidores si no quieren que los paquetes se queden mucho tiempo bloqueados ya que, como se ha visto, no se soporta de manera óptima el envío de varios paquetes al tener que depender siempre de un paquete anterior.

Que HTTP/1.1 tenga que establecer varias conexiones supone un problema de rendimiento puesto que hay que emplear tiempo en establecer cada una de esas conexiones, las cuales comenzarán una fase de *slow start*. Además, si esas conexiones fueran cifradas, cada una de ellas tendría que emplear tiempo para negociar el cifrado. Esto también implicaría un problema y es que si la red sufriera problemas de rendimiento y provocara retrasos a los paquetes, al haber varias conexiones iniciadas simultáneamente, serían varios los paquetes que se verían afectados por esos retrasos.

HTTP/2 soluciona esto mediante el uso de una única conexión TCP sobre la que viajan tanto las solicitudes como las respuestas haciendo uso de diferentes flujos de datos, los cuales pueden tener una prioridad asignada. Esto significa que el ancho de banda de la red se aprovecha de manera más eficiente ya que sólo hay que establecer una conexión TCP y, en caso de retrasos, solo se ve afectada una conexión. También se mejora el transporte de los datos de cabecera utilizando HPACK [17], que es un protocolo utilizado para comprimir las cabeceras de las peticiones. Además, establece un sistema de priorización de solicitudes lo que mejora aún más el rendimiento del protocolo. Y por último, otra de las mejoras destacables del protocolo es el uso de formato binario para la trasmisión de mensajes.

4.1.2. Características básicas del protocolo

4.1.2.1. Inicio HTTP/2

HTTP/2 es un protocolo que funciona a nivel aplicación y que utiliza TCP como nivel de transporte.

Las URIs de HTTP/2 son iguales que las del protocolo HTTP/1.1 y los puertos http y https son los mismos que utilizaba la versión anterior, 80 y 443 respectivamente. Esto está hecho con el objetivo de no tener que cambiar toda la infraestructura de Internet y que puedan convivir ambos protocolos. De este modo, si las URIs y los puertos son los mismos, será el propio protocolo el que tendrá que identificar si el servidor al que se le está haciendo la petición de recursos soporta o no el nuevo protocolo.

Para saber si el servidor soporta o no la versión HTTP/2 se utiliza un mecanismo muy sencillo. Por una parte, si el cliente va a hacer uso de conexiones no cifradas y va a atacar a una URL del tipo "http", lo único que tiene que hacer es mandar una solicitud tal y como lo haría para HTTP/1.1 pero añadiendo una cabecera llamada `Upgrade`. En esa nueva cabecera se pueden incluir dos valores diferentes. Por un lado "h2c" que significará que la conexión de HTTP/2 no irá cifrada o, "h2" lo cual hará que se comience la negociación de una conexión segura y se procesará utilizando el protocolo ALPN (*Application-Layer Protocol Negotiation*) [19].Este protocolo es una extensión de TLS (*Transport Layer Security*) que permite negociar qué protocolo debe utilizarse en las conexiones seguras a nivel de aplicación, lo cual hace que sea mucho más eficiente que una negociación a nivel de enlace.

Si el servidor soportara HTTP/2 sin conexión cifrada contestaría con un mensaje "HTTP/1.1 101 Switching Protocols".

Si por otro lado el cliente atacara a una URL del tipo "https", se haría uso del protocolo ALPN. Esto consiste en incluir la información de los protocolos que se quieren utilizar a la hora de negociar la conexión TLS. Cuando el cliente envía el primer paquete para iniciar la negociación con el servidor, le indica los protocolos que quiere utilizar. El servidor, cuando contesta a ese primer paquete, indica al cliente el protocolo que va a utilizar. Se puede comprobar en el ejemplo obtenido al acceder a la web de Google observando las figuras 4.1 y 4.2.

```
Extension: Application Layer Protocol Negotiation
  Type: Application Layer Protocol Negotiation (0x0010)
  Length: 14
  ALPN Extension Length: 12
  ALPN Protocol
      ALPN string length: 2
      ALPN Next Protocol: h2
      ALPN string length: 8
      ALPN Next Protocol: http/1.1
```

Figura 4.1: ALPN-Inicio de negociación del cliente

```
Extension: Application Layer Protocol Negotiation
  Type: Application Layer Protocol Negotiation (0x0010)
  Length: 5
  ALPN Extension Length: 3
  ALPN Protocol
   ALPN string length: 2
   ALPN Next Protocol: h2
```

Figura 4.2: ALPN-Contestación del servidor

Tras esto, tanto si el protocolo funcionara utilizando TLS como si funcionara sin cifrar, el servidor y el cliente deberán enviar un mensaje antes de que se establezca la conexión cuyo objetivo es confirmar que el protocolo que se va a usar es HTTP/2. Este *frame* comienza con una secuencia de 24 octetos con el string:

```
PRI * HTTP/2.0\r\n\r\nSM\n\r\n
```

Cuando el receptor procese el contenido del *frame* deberá enviar inmediatamente otro *frame* con el *flag* `ACK` activado que, además, podrá contener cabeceras de configuración adicionales de manera que se reducirá el tráfico de paquetes que circulan por la red.

Una vez se haya establecido la sesión HTTP/2, al mensaje de la primera solicitud se le asignará el identificador de flujo "1". Y ese mismo flujo se utilizará para el envío de la primera respuesta por parte del servidor.

4.1.2.2. Frames HTTP/2

En primer lugar, hay que mencionar que cada paquete en HTTP/2 puede estar compuesto por varios *frames* los cuales tienen una cabecera fija de 9 octetos seguida por un *payload* variable. Los campos de la cabecera son:

- `Length`: Es la longitud del campo de datos del *frame*, es decir, no se incluyen los 9 octetos de cabecera. El valor de este campo se expresa como un entero sin signo de 24 bits.
- `Type`: Expresa el tipo de *frame* y, por tanto, el formato del *frame*. Hay una lista de valores fijados, cualquier *frame* que tenga este campo con un valor desconocido debe ser descartado.
- `Flags`: Dependen del tipo de *frame* especificado en el campo anterior, en caso de que el tipo de *frame* no necesite usar ningún *flag*, este campo se dejará con valor (0x0).
- `Stream Identifier`: Expresa el identificador del flujo de datos como un entero sin signo de 31 bits.
- `Payload`: Es el campo de datos y dependerá del tipo de *frame*.

El tamaño máximo de un *frame* en HTTP/2 está limitado por el tamaño máximo indicado en la opción `SETTINGS_MAX_FRAME_SIZE` y el tamaño mínimo que se debe poder recibir y procesar es el equivalente a los 9 octetos de la cabecera más 2^{14} octetos.

Por último, las cabeceras de las peticiones en esta versión de HTTP pueden ser comprimidas. Para esta tarea se ha ideado un protocolo llamado HPACK el cual explicaré más a fondo en el apartado 4.2. Como breve introducción, mediante este protocolo se puede comprimir la cabecera de los paquetes eliminando las cabeceras redundantes. Por ejemplo, en HTTP/1.1 si hacemos 20 peticiones a un servidor web las 20 respuestas podrían venir con las mismas *cookies*, con HPACK esa información redundante desaparecería. Para finalizar, cabe destacar que el protocolo se ha diseñado de forma específica para HTTP/2 con el objetivo de eliminar posibles vulnerabilidades como BREACH [5] o CRIME [18].

4.1.2.3. Principales tipos de Frames HTTP/2

Para explicar los diferentes tipos de *frames* existentes en HTTP/2 me basaré en la imagen 4.3 en la cual se puede observar un sencillo intercambio de paquetes entre cliente y servidor.

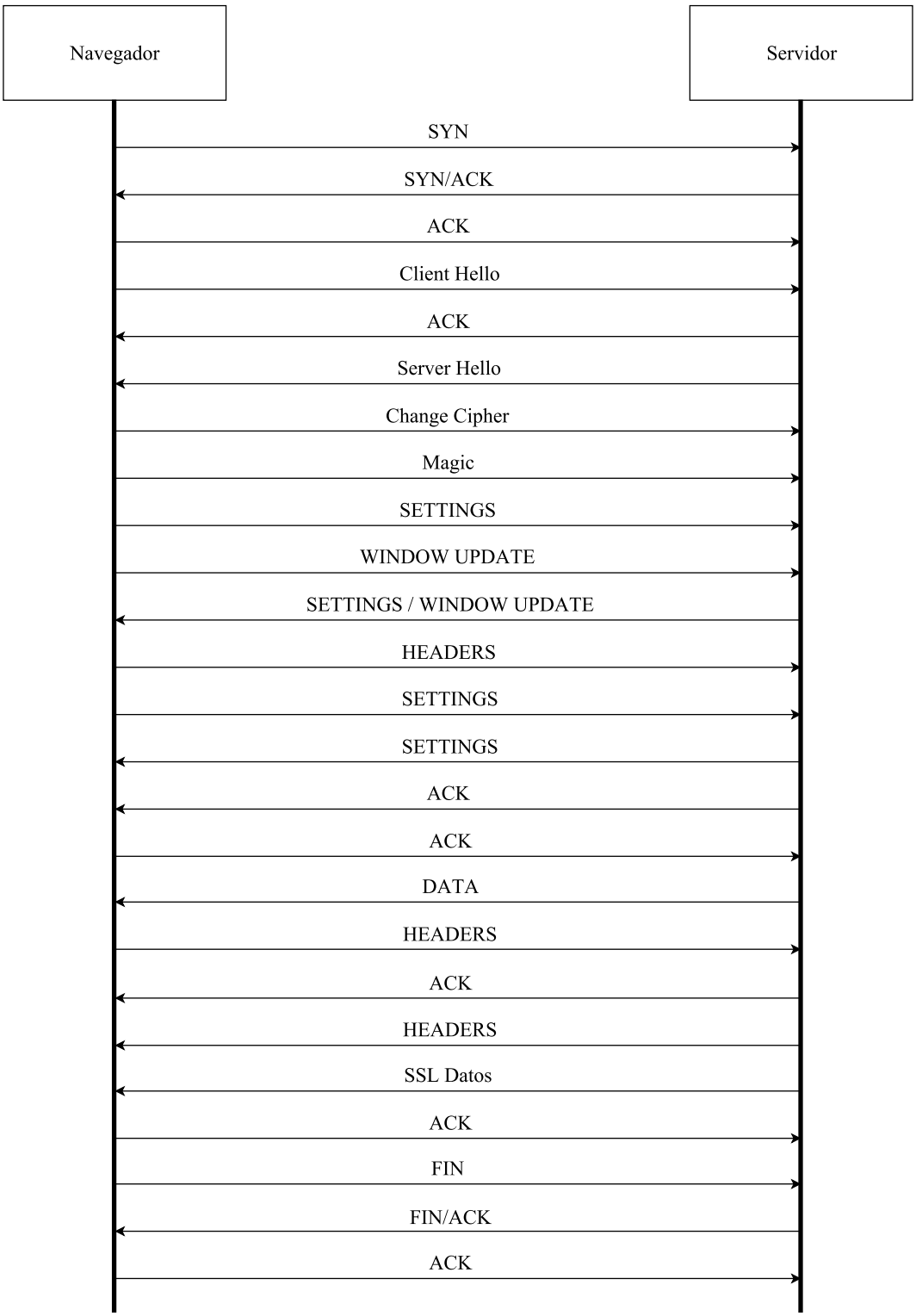


Figura 4.3: Intercambio de paquetes en HTTP2

El primer *frame* HTTP/2 que aparece en la imagen es de tipo `SETTINGS` y se identifica con el valor `(0x4)` en el campo `type` de la cabecera de los paquetes HTTP/2. Sirve para configurar los parámetros que afectan a la comunicación entre los puntos finales.

Además, también será utilizado cuando se haya recibido alguna configuración de parámetros a modo de acuse de recibo el cual deberá llevar activo el *flag* `ACK`.

Este *frame* solo puede ser enviado por los puntos finales y generalmente se envía al comienzo de una conexión. Sin embargo, también es posible que durante el transcurso de la conexión aparezcan nuevos *frames* actualizando cualquiera de los parámetros previamente establecidos.

Este tipo de *frames* aplica a una conexión, nunca a un flujo y el identificador de flujo debe ser `0x0`.

El interior del *frame* `SETTINGS` se compone de cero o más parámetros los cuales están formados por un identificador de 16 bits y un valor de 32 bits. Los parámetros definidos son:

- `SETTINGS_HEADER_TABLE_SIZE (0x1)`: Permite al transmisor del *frame* informar del tamaño máximo de la tabla utilizada para la descompresión de las cabeceras con HPACK.
- `SETTINGS_ENABLE_PUSH (0x2)`: Esta opción se utiliza para deshabilitar el envío de *frames* utilizando la nueva característica denominada server push, la cual explico más adelante. Si el valor es cero, significa que está deshabilitado.
- `SETTINGS_MAX_CONCURRENT_STREAMS (0x3)`: Indica el número de flujos de datos máximos permitidos por conexión. Se recomienda que este valor no sea menor de 100 y, salvo que se especifique un valor concreto en esta opción, el número de flujos es ilimitado, es decir, hasta que se acabe el número de identificadores disponibles para asignar.
- `SETTINGS_INITIAL_WINDOWS_SIZE (0x4)`: Indica el tamaño de ventana inicial que utilizará el transmisor.
- `SETTINGS_MAX_FRAME_SIZE (0x5)`: Indica el tamaño máximo del *payload* que se puede trasmitir. Por defecto, este valor es 2^{14} y como máximo puede tener un valor de $2^{24} - 1$ octetos.

- `SETTINGS_MAX_HEADER_LIST_SIZE (0x6)`: Esta opción sirve para avisar del tamaño máximo de la lista de cabeceras. El cálculo de este valor aplica al tamaño de las cabeceras antes de ser comprimidas.

A continuación se adjunta un ejemplo de una captura realizada con wireshark:

```
Stream: SETTINGS, Stream ID: 0, Length 24
    Length: 24
    Type: SETTINGS (4)
    Flags: 0x00
        .... ...0 = ACK: False
        0000 000. = Unused: 0x00
    0... .... .... .... .... .... .... .... = Reserved: 0x00000000
     .000 0000 0000 0000 0000 0000 0000 0000 = Stream Identifier: 0
    Settings - Max concurrent streams : 100
    Settings - Initial Windows size : 1048576
    Settings - Max frame size : 16384
    Settings - Max header list size : 16384
```

Figura 4.4: Frame SETTINGS

El segundo *frame* HTTP/2 que aparece en la imagen es el de tipo `WINDOW_UPDATE` y se idenfifica con el valor `(0x8)` en el campo `type` de la cabecera de los paquetes HTTP/2. Sirve para implementar el control de flujo a nivel HTTP/2, no a nivel TCP como sería habitual en la pila de protocolos TCP/IP. Este control de flujo se explicará en la sección 4.1.3.1.

Los campos por los que están formados los *frames* `WINDOW_UPDATE` son:

- `R`: Un bit reservado
- `Window Size Increment`: Este valor indica lo que se va a incrementar el valor del tamaño de la ventana respecto al valor que había antes. Es decir, si la ventana fuera de 1024 octetos, y el incremento 2048 octetos, la nueva ventana sería de 3072 octetos.

Además, este *frame* lleva un identificador para saber si el control de flujo que se actualiza aplica a la conexión o a un flujo de datos. Si el identificador fuera igual a cero, se estaría actualizando la ventana de la conexión, en caso contrario se estará actualizando la ventana del flujo al que se haga referencia.

A continuación, adjunto dos imágenes, una en la que se actualiza el tamaño de la ventana de conexión(identificador de flujo a cero) y otra en el que se actualiza el tamaño de uno de los flujos.

```
Stream: WINDOW_UPDATE, Stream ID: 0, Length 4
    Length: 4
    Type: WINDOW_UPDATE (8)
    Flags: 0x00
        0000 0000 = Unused: 0x00
    0... .... .... .... .... .... .... .... = Reserved: 0x00000000
    .000 0000 0000 0000 0000 0000 0000 0000 = Stream Identifier: 0
    0... .... .... .... .... .... .... .... = Reserved: 0x00000000
    .000 0000 0000 1111 0000 0000 0000 0001 = Window Size Increment: 983041
```

Figura 4.5: Frame WINDOW_UPDATE conexión

```
Stream: WINDOW_UPDATE, Stream ID: 3, Length 4
    Length: 4
    Type: WINDOW_UPDATE (8)
    Flags: 0x00
        0000 0000 = Unused: 0x00
    0... .... .... .... .... .... .... .... = Reserved: 0x00000000
    .000 0000 0000 0000 0000 0000 0000 0011 = Stream Identifier: 3
    0... .... .... .... .... .... .... .... = Reserved: 0x00000000
    .000 0000 1001 1111 0000 0000 0000 0000 = Window Size Increment: 10420224
```

Figura 4.6: Frame WINDOW_UPDATE-2 flujo

El tercer *frame* en aparecer es el denominado `HEADERS` el cual se identifica con el valor `(0x1)` en el campo type de la cabecera de los paquetes HTTP/2. Se encarga de llevar las cabeceras HTTP así como de iniciar nuevos flujos de datos HTTP/2.

Es importante saber que estos *frames* deben ir asociados siempre a un flujo de datos.

Los *frames* `HEADERS` están formados por los siguientes campos:

- `Pad Length`: Campo de 8 bits que contiene la longitud del *padding* en octetos. Este campo no es de obligado cumplimiento y únicamente se completará en el caso de que la cabecera tenga el *flag* PADDED activo.

- `E`: Es un *flag* de un solo bit que indica si la dependencia de flujos es exclusiva. Este campo sólo estará presente si el *flag* PRIORITY está activo en la cabecera.

- `Stream Dependency`: Es un entero de 31 bits que contiene el identificador del flujo del cual se está dependiendo lo que implica que no se comenzará a intercambiar paquetes hasta que termine el flujo del que depende. Este campo sólo estará presente si el *flag* PRIORITY está activo en la cabecera.

- `Weight`: Es un entero sin signo de 8 bits que pondera la prioridad del flujo de datos. Este campo sólo estará presente si el *flag* PRIORITY está activo en la cabecera.

- `Header Block Fragment`: Contiene un fragmento de cabeceras.

- `Padding`: El campo *padding* contiene datos sin información útil para el protocolo.

Los campos que dependían del frame PRIORITY se explicarán prestando especial atención en la sección 4.1.3.2.

Además, el *frame* dispone de varios *flags* que puede utilizar aunque los más importantes son:

- `END_STREAM(0x1)`: Cuando está activo, indica que este fragmento de cabeceras será el último que va a ser enviado en ese flujo de datos. Si un *frame* de tipo `HEADERS` no tiene este *flag* activo, el siguiente *frame* que se envíe debe ser de tipo `CONTINUATION` para el mismo flujo de datos. Este se tratará como si fuera la continuación del *frame* de cabeceras.

- `END_HEADERS(0x4)`: Cuando está activo significa que contiene un bloque de cabeceras completo y no se complementará con ningún *frame* `CONTINUATION`. Si este *flag* no estuviera activo, el *frame* debería ir precedido por un *frame* de tipo `CONTINUATION`.

A continuación se adjunta una imagen con un ejemplo de un *frame* `HEADERS`:

```
Stream: HEADERS, Stream ID: 1, Length 246
    Length: 246
    Type: HEADERS (1)
    Flags: 0x25
        .... ...1 = End Stream: True
        .... .1.. = End Headers: True
        .... 0... = Padded: False
        ..1. .... = Priority: True
        00.0 ..0. = Unused: 0x00
    0... .... .... .... .... .... .... .... = Reserved: 0x00000000
    .000 0000 0000 0000 0000 0000 0000 0001 = Stream Identifier: 1
    [Pad Length: 0]
    1... .... .... .... .... .... .... .... = Exclusive: True
    .000 0000 0000 0000 0000 0000 0000 0000 = Stream Dependency: 0
    Weight: 255
    [Weight real: 256]
    Header Block Fragment: 82418cf1e3c2f28c858ce7eab90f4f87844092b6b9ac1c85...
    [Header Length: 461]
    [Header Count: 10]
    Header: :method: GET
    Header: :authority: www.facebook.com
    Header: :scheme: https
    Header: :path: /
    Header: upgrade-insecure-requests: 1
    Header: user-agent: Mozilla/5.0 (Windows NT 6.3; WOW64) AppleWebKit/537.36 (
    Header: accept: text/html,application/xhtml+xml,application/xml;q=0.9,
    image/webp,*/*;q=0.8
    Header: accept-encoding: gzip, deflate, sdch, br
    Header: accept-language: es-ES,es;q=0.8
    Header: cookie: datr=XpOKVpuTpDt_xe6YNX1PCvmh
    Padding: <MISSING>
```

Figura 4.7: Frame HEADERS

Por último, aparece el *frame* `DATA` el cual se identificará con el valor `(0x0)` en el campo `type` de la cabecera de los paquetes HTTP/2 y sirve para enviar los datos propiamente dichos.

Este tipo de *frames* pueden incluir bits de relleno que se utilizan como protección de los datos de manera que, en caso de sufrir ataques, estos

ataques no conocen el tamaño real de los *frames*.

Es muy importante mencionar que, al igual que los *frames* `HEADERS`, estos *frames* deben ir asociados siempre a algún flujo de datos.

Los *frames* `DATA` están formados por los siguientes campos:

- `Pad Length`: Campo de 8 bits que contiene la longitud del *padding* en octetos. No es necesario completar este campo y únicamente se rellenará en el caso de que la cabecera tenga el *flag* `PADDED` activo.
- `Data`: Este campo contiene los datos de aplicación. El tamaño de este campo es variable.
- `Padding`: El campo *padding* contiene datos sin información útil para el protocolo.

Los *frames* `DATA` están sujetos a un control de flujo que proporciona HTTP/2, el cual explicaré en la sección 4.1.3.1. Esto significa que solo pueden ser enviados si el flujo se encuentra en el estado `open` o `half-closed (remote)`. Todo *frame* de datos se incluye en el control de flujo, incluso el *padding*.

A continuación se muestra un ejemplo de un *frame* de tipo `DATA`:

```
Stream: DATA, Stream ID: 1, Length 11371
    Length: 11371
    Type: DATA (0)
    Flags: 0x00
        .... ...0 = End Stream: False
        .... 0... = Padded: False
        0000 .00. = Unused: 0x00
    0... .... .... .... .... .... .... .... = Reserved: 0x00000000
     .000 0000 0000 0000 0000 0000 0000 0001 = Stream Identifier: 1
    [Pad Length: 0]
    Data: 1b1d6283b9088452017fa7c316ac94a1157df328e7fcaed7...
    Padding: <MISSING>
```

Figura 4.8: Frame DATA

4.1.2.4. Otros frames HTTP/2

A continuación se van a citar junto a una breve explicación los otros *frames* que aparecen en este protocolo pero que, sin embargo, no son tan importantes y no se veían reflejados en la captura 4.3.

`PRIORITY`: Este tipo de *frames* se identifican con `(0x2)` en el campo `type` de la cabecera de los paquetes HTTP/2. Sirve para especificar la prioridad de un flujo de datos y siempre debe ir asociado a un flujo de datos.

El *frame* está formado por los siguientes campos:

- `E`: Es un *flag* de un solo bit que indica si la dependencia de flujos es exclusiva. Este campo sólo estará presente si el *flag* `PRIORITY` está activo en la cabecera.
- `Stream Dependency`: Es un entero de 31 bits que contiene el identificador del flujo del cual se está dependiendo.Este campo sólo estará presente si el *flag* `PRIORITY` está activo en la cabecera.
- `Weight`: Es un entero sin signo de 8 bits que pondera la prioridad del flujo de datos. Este campo sólo estará presente si el *flag* `PRIORITY` está activo en la cabecera.

`PUSH_PROMISE`: Este tipo de *frames* se identifican como `(0x5)` en el campo `type` de la cabecera de los paquetes HTTP/2. Sirve para notificar al cliente, los *frames* que el servidor va a intentar enviarle antes de que este los solicite. En el apartado 4.1.3.3, explico más a fondo el funcionamiento de este *frame*.
Este *frame* está compuesto por los siguientes campos:

- `Pad Length`: Campo de 8 bits que contiene la longitud del *padding* en octetos. Este campo no es de obligado cumplimiento y únicamente se completará en el caso de que la cabecera tenga el *flag* PADDED activo.
- `R`: Un bit reservado
- `Promised Stream ID`: Es un campo de 31 bits que sirve para indicar el identificador del flujo de datos reservado por el servidor para enviar los *frames* al cliente.
- `Header Block Fragment`.
- `Padding`: El campo *padding* contiene datos que no contienen información útil para el protocolo.

`CONTINUATION`: Este tipo de *frames* se identifican como `(0x9)` en el campo `type` de la cabecera de los paquetes HTTP/2. Sirve para continuar enviando fragmentos del bloque de cabeceras, además del fragmento recibido en el *frame* `HEADERS` anterior.

Los *frames* `CONTINUATION` deben ir asociados a un flujo de datos y siempre deben estar precedidos por un *frame* `HEADERS`, `PUSH_PROMISE` o `CONTINUATION` sin el *flag* `END_HEADERS` activo.

Únicamente está compuesto por un bloque de cabeceras y define el *flag* `END_HEADERS` que utilizará para indicar que el bloque de cabeceras termina en este *frame*. Si este *flag* no estuviera activo, a continuación debería ir otro bloque `CONTINUATION`.

`RST_STREAM`: Este tipo de *frames* se identifican como `(0x3)` en el campo `type` de la cabecera de los paquetes HTTP/2. Sirve para terminar de forma inmediata un flujo por lo que, al igual que pasa con otros *frames*, siempre debe ir asociado a un flujo de datos. La cancelación puede hacerse o bien mediante una solicitud o bien porque haya ocurrido algún error en ese flujo.

Estos *frames* solo incluyen el campo `Error Code` que es un campo entero de 32 bits en el que se especifica el código de error. No se define ningún *flag*. Los errores definidos en el protocolo se pueden encontrar en la sección 7 de la RFC de HTTP/2.

Este *frame* termina completamente el flujo dejándolo en el estado `closed`. Cuando el receptor reciba el *frame*, no podrá enviar ningún otro a excepción de *frames* `PRIORITY`.

`PING`: Este tipo de *frames* se identifican como `(0x6)` en el campo `type` de la cabecera de los paquetes HTTP/2. Sirve para comprobar que la conexión entre dos puntos funciona correctamente y pueden ser enviados tanto por el cliente como por el servidor. Como es lógico, este *frame* deberá ir asociado siempre a la conexión HTTP/2 y nunca a un flujo en particular.

El *frame* `PING` define un único *flag* denominado `ACK` que cuando está a cero indica que es una respuesta.

`GOAWAY`: Este tipo de *frames* se identifican como `(0x7)` en el campo `type` de la cabecera de los paquetes HTTP/2. Sirve para iniciar el proceso de fin de una conexión o para indicar que se ha producido un error muy grave en

la conexión. Si se recibe este mensaje implica que no se pueden enviar ni recibir nuevos *frames*, pero sí se pueden terminar de procesar los que ya se hubieran recibido.

Este *frame* no tiene definido ningún *flag*, y está formado por los campos:

- `R`: Un bit reservado.
- `Last-Stream-ID`: Sirve para indicar el identificador del último stream sobre el que se ha realizado alguna acción como enviar, recibir o procesar *frames*. Los *frames* recibidos por flujos con menor identificador deben terminar de procesarse, pero los que vengan de flujos con mayor identificador deben ser descartados de forma inmediata.
- `Error Code`: Código que indica el motivo por el cual se cierra la conexión. Utiliza los mismos códigos de error que el *frame* de tipo `RST_STREAM`.
- `Additional Debug Data`: Opcionalmente, se puede incluir este campo con el objetivo de utilizarlo con fines de depuración para facilitar el diagnóstico de problemas.

4.1.2.5. Tipos de errores

HTTP/2 permite manejar dos tipos de errores. Aquellos que afectan a toda la conexión, los cuales serán denominados errores de conexión y los que afectan a un único flujo de datos, que se llamarán errores de flujo.

Los errores de conexión se lanzan para prevenir que se sigan produciendo errores que puedan dar lugar a datos corruptos por problemas en la conexión. Cuando se detecta un error de este tipo, se debe enviar un *frame* de tipo `GOAWAY` que incluye un código de error para explicar el motivo del cierre de la conexión.

Cuando se produce un error de flujo únicamente se cierra el flujo afectado, por lo que el impacto sobre la conexión es mínimo. Al detectar un error de este tipo, se envía un *frame* `RST_STREAM` que incluye un código de error para explicar el motivo del cierre del flujo.

4.1.3. Nuevas características del protocolo

4.1.3.1. Control de flujo en HTTP/2

Debido a que HTTP/2 se apoya sobre una única conexión TCP, el mecanismo de control de flujo de TCP no es lo suficientemente eficiente ya que no sólo hay que controlar el tamaño máximo de datos que podemos recibir en la conexión sino que al existir varios flujos de datos, hay que asegurarse de que el receptor no se vaya a quedar sin espacio para procesar cada uno de esos flujos. Para tratar de limitar ese problema, y que eso no afecte tanto al intercambio de paquetes en HTTP/2, se ha ideado un mecanismo de control de flujo propio para este nuevo protocolo.

Antes de explicar lo que es el control de flujo en HTTP/2 conviene explicar lo que se considera un flujo HTTP/2.

Un flujo en HTTP/2 es una secuencia de *frames* independientes y bidireccionales que se intercambian entre cliente y servidor. Estos flujos tienen varias características aunque voy a citar solamente alguna de las más importantes:

- Una conexión HTTP/2 puede contener varios flujos HTTP/2 abiertos de forma simultanea.
- Un mismo flujo puede ser utilizado tanto por el cliente como por el servidor.
- Los flujos pueden cerrarse en cualquier momento por el cliente o por el servidor independientemente de quien lo originara.
- Los flujos se pueden diferenciar gracias a un identificador de flujo el cual será "1" para el primer mensaje intercambiado bajo HTTP/2. Cualquier flujo originado tras el primer mensaje deberá tener un identificador superior a "1".

Los flujos HTTP/2 pueden encontrarse en varios estados diferentes los cuales se pueden comprobar en la sección 5.1 de la RFC de HTTP/2.

Para abrir nuevos flujos de datos se utilizan *frames* de tipo `HEADERS` y si sólo se quiere reservar para utilizarlo en un futuro, se hará con *frames* de tipo `PUSH_PROMISE`.

Los identificadores no pueden ser reutilizados al originar nuevos flujos de datos, es decir, si ya ha existido durante el transcurso de la conexión

un flujo con identificador "20", nunca podrá volver a existir algún flujo con ese mismo identificador. Si estos números identificadores se acabaran, habría que establecer una nueva conexión TCP. En el caso de que ocurra, el servidor avisará al cliente con un *frame* de tipo `GOAWAY` para que establezca la nueva conexión.

Características del control de flujo de HTTP/2

Una vez ya se sabe lo que es un flujo de datos, voy a citar alguna de las características más importantes del control de flujo en HTTP/2:

- El control de flujo siempre se realizará entre puntos finales.
- El control de flujo sólo aplica sobre los *frames* de tipo `DATA`. El resto de *frames* no se tienen en cuenta a la hora de calcular el tamaño de la ventana. Esto asegura que los *frames* importantes, como por ejemplo las cabeceras o las opciones de la conexión, nunca se queden bloqueados debido al control de flujo.
- El control de flujo se lleva a cabo por el receptor. Este receptor puede tener una ventana de tamaño mayor para un flujo que para otro y el emisor debe respetarlo.
- Por defecto, la ventana se inicializa a 65.535 octetos.
- El control de flujo que realiza HTTP/2 no puede ser deshabilitado.

El control de flujo se define para evitar que los participantes de la conexión tengan problemas de memoria y problemas debidos a su velocidad de subida/descarga de datos.

El funcionamiento es muy sencillo, los receptores de paquetes pueden avisar al servidor mediante *frames* de tipo `WINDOW_UPDATE` del tamaño máximo de datos en octetos que quieren recibir. Esta ventana aplicará directamente sobre el flujo de datos sobre el que vaya el *frame* `WINDOW_UPDATE`. Esto es muy útil puesto que el cliente puede controlar cuantos paquetes quiere recibir en cada momento, si su buffer de recepción estuviera saturado, podría avisar al servidor de que le envíe menos datos. O si por ejemplo se necesitase que por un flujo de datos se enviaran todos los datos porque fueran muy importantes se podría poner el tamaño de ventana del resto de flujos a 0, para que el flujo elegido tuviera todo el ancho de banda. No obstante, la gran ventaja

del control de flujo en HTTP/2 es que deja a elección de los desarrolladores gestionar el flujo de datos de la manera que ellos consideren más eficiente, es decir, los dos supuestos anteriores son casos que podrían querer controlar los desarrolladores de servidores HTTP/2, pero podrían gestionar el flujo como quisieran para lo que necesitasen.

4.1.3.2. Priorización de flujos

El objetivo de la priorización de flujos es que los clientes/servidores puedan especificar cómo quieren recibir esos *frames*. Por ejemplo, si se necesita un *frame* importante y la ventana se está agotando, se puede priorizar el flujo por el que debe llegar ese *frame* para que este sea recibido antes de que la ventana se agote. O si se quisiera cancelar una conexión, se podría priorizar el flujo por el que se va a enviar el aviso de que se cancela la conexión.

No obstante, es importante recalcar que esta prioridad no obliga a procesar un flujo antes que el resto, simplemente es una recomendación que podría no ser tenida en cuenta por el cliente/servidor.

Para configurar la priorización de flujos en HTTP/2 se hace uso de los *frames* de tipo `HEADERS` y `PRIORITY`. El *frame* `HEADERS` sirve para configurar que el flujo de datos haga uso de la nueva funcionalidad de priorización de flujos y el *frame* `PRIORITY` sirve para cambiar la prioridad de un flujo en cualquier momento.

Las prioridades en HTTP/2 se controlan mediante "pesos" y dependencias de otros flujos. El peso de la prioridad que se le de a un flujo puede variar desde 1 hasta 256 y en caso de haber varios flujos se establecerá una proporción de datos a recibir de un flujo respecto al resto de flujos.

Pongamos un ejemplo:

- Hay tres flujos: A, B y C.
- El flujo A tiene un peso de 10 y depende del flujo C.
- El flujo B tiene un peso de 7 y depende del flujo C.
- El flujo C tiene un peso de 15.

En el caso del ejemplo el flujo C debería de terminar de recibir todos los datos en primer lugar. Una vez el flujo C ha terminado, los flujos A y B

pueden comenzar a recibir los datos y lo harán según la proporción debida a sus pesos.

En primer lugar, se suman todos los pesos: $10 + 7 = 17$ Tras ello, se calcula la parte proporcional de recursos que va a poder utilizar cada flujo de datos: $A = 10/17$ $B = 7/17$

4.1.3.3. Server push

Esta es una de las nuevas características del protocolo y ofrece al servidor la posibilidad de enviar respuestas al cliente a pesar de que este aún no las haya solicitado. Por ejemplo, si el cliente quiere entrar a una web que tiene varias imágenes lo que permite esta característica es que el servidor, en lugar de esperar a que el cliente le solicite las imágenes, envíe junto con el archivo `html` todas las imágenes asociadas al mismo. De modo que se ahorra el tiempo que se emplearía en todo ese intercambio de solicitudes-respuestas.

Esta característica no es obligatoria y puede ser deshabilitada por el cliente configurando la opción `SETTINGS_ENABLE_PUSH` a cero.

Las respuestas enviadas de esta manera pueden ser almacenadas en caché por parte del cliente por lo que aumentaría aún más el rendimiento de la navegación web al almacenar esos recursos de forma local de cara a futuras peticiones.

Solicitudes push

Hacer `server push` es equivalente a responder a una solicitud, sin embargo, como hay que anteponerse a ellas, es necesario crear una solicitud ficticia que va a enviar el propio servidor antes de transmitir los *frames* de datos. Esta solicitud debe incluir un bloque de cabeceras la cual llevará el identificador de flujo por el que se deben enviar los *frames* a los que se hacen `push`.

Este *frame* con la solicitud ficticia es de tipo `PUSH_PROMISE` y solo puede ser enviado por un servidor, en ningún caso por un cliente. El envío de este *frame* crea un nuevo flujo de datos por el que se enviarán el resto de los recursos que el servidor entiende que el cliente va a querer aunque éste no se los haya solicitado todavía.

Respuestas push

Tras enviar el *frame* `PUSH_PROMISE`, el servidor puede empezar a enviar las respuestas por el flujo de datos creado anteriormente. Cuando el cliente recibe el mensaje `PUSH_PROMISE` puede decidir rechazar el envío de esos mensajes o aceptarlo. En caso de aceptarlo, no podrá enviar ninguna solicitud hasta que el flujo de datos habilitado para el envío `push` se cierre.

Siempre que el cliente reciba una respuesta `push`, debe comprobar que el servidor está autorizado. Por ejemplo, si el servidor está autorizado para el envío de push para la dirección `Halffter.gast.it.uc3m.es` no podrá enviar mensajes push para `Halffter.gast.it.uc3m.es/images` puesto que, aunque pertenece al mismo dominio, la ruta no es exactamente igual.

4.1.3.4. Protocolo binario

Este protocolo trae como otra novedad importante que sus paquetes viajarán por la red en formato binario. Es una novedad interesante puesto que hace que el envío de los paquetes sea más eficiente y provoque menos errores. Esto se debe a que los datos van mucho más compactos y, por tanto, ocupan menos espacio. Como desventaja tiene que los paquetes no son interpretables al igual que puede serlo un paquete ASCII, pero con plugins como los que trae instalados Wireshark no hay ningún problema para leer su contenido.

4.1.4. Otras características del protocolo

4.1.4.1. Intercambio de mensajes HTTP

Intercambio solicitud/respuesta

HTTP/2 se diseñó con el objetivo de que fuera lo más compatible posible con los protocolos HTTP anteriores. Esto quiere decir que, a nivel semántico, el protocolo apenas ha sufrido cambios conservando la semántica tanto de las solicitudes como de las respuestas.

La novedad más importante respecto a HTTP/1.1 es que cuando el cliente envía una solicitud por un flujo de datos, la respuesta por parte del servidor debe enviarse por el mismo flujo de datos. En HTTP/1.1 al no existir el concepto de flujo, todas las respuestas se devuelven por la conexión por la que se haya enviado la solicitud.

Este intercambio de solicitudes y respuestas se hará siempre apoyándose

en los *frames* `HEADERS` los cuales tienen un campo en el que se puede introducir un bloque de cabeceras HTTP.

Es posible que con un único *frame* de tipo `HEADERS` no sea suficiente para enviar todas las cabeceras porque se excediera el tamaño máximo del *frame*. En ese caso se podrá hacer uso del *frame* `CONTINUATION` el cual sirve para enviar las cabeceras adicionales que no se hubieran mandado.

El último *frame* `HEADERS` o `CONTINUATION` que contenga cabeceras debe llevar activo el *flag* `END_HEADERS` para avisar de que ya no hay más cabeceras.

Cabeceras HTTP

Las cabeceras HTTP son cadenas de caracteres codificadas en ASCII y todos los caracteres se convierten a minúsculas antes de codificar las cabeceras en HTTP/2.

En caso de que algún proxy tuviera que transformar un paquete HTTP/1.1 a HTTP/2 es imprescindible que elimine las cabeceras de conexión (`keep-alive`, `proxy-connection`, `transfer-encoding` y `upgrade`).

Adjunto a continuación una lista de las pseudo-cabeceras más importantes de HTTP/1.1 que siguen estando presentes en este protocolo:

- `:method` : Incluye el método HTTP(`GET`, `POST`, `CONNECT`, `TRACE`, etc)
- :scheme
- :authority
- :path : Este campo nunca debe estar vacío para URIs http o https.

Todas las solicitudes HTTP/2 deberán tener un valor válido para las pseudo cabeceras `:method`,`:scheme` y `:path`.

Para las respuestas, solo se define como obligatoria la pseudo cabecera `:status`.

Para la gestión de las cookies todo sigue igual que en HTTP/1.1 salvo que ahora se crean varios campos de cabecera en los que se incluirá una cookie por campo, mientras que antes era todo una lista de cookies separadas

por punto y coma.

A continuación adjunto una imagen con las cabeceras que aparecen en un *frame* `HEADERS` en el que se puede observar las diferencias a la hora de gestionar cookies y las cabeceras obligatorias citadas anteriormente:

```
   [Header Length: 1254]
[Header Count: 16]
Header: :method: GET
Header: :authority: www.google.es
Header: :scheme: https
Header: :path: /complete/search?client=chrome-omni&gs_ri=chrome-ext-ansg&x
    ssi=t&q=fa&oit=1&cp=2&pgcl=4&gs_rn=42&psi=cVsxG7h2Jf1fjXXG&
sugkey=AIzaSyBOti4mM-6x9WDnZIjIeyEU21OpBXqWBgw
Header: x-client-data: CIu2yQEIo7bJAQ==
Header: user-agent: Mozilla/5.0 (Windows NT 6.3; WOW64) AppleWebKit/537.36
(KHTML, like Gecko) Chrome/51.0.2704.103 Safari/537.36
Header: accept-encoding: gzip, deflate, sdch, br
Header: accept-language: es-ES,es;q=0.8
Header: cookie: SID=YwPkqZQnbPi-5dLExGU5d4BTJp0ILE855-RLvC
pXDcYiGcW4hjgSqD9v8A1DSjcAydcj0A.
Header: cookie: HSID=Ahg24wX9oOoKzSJUa
Header: cookie: SSID=ABLQ3ODK0sTUMXmD_
Header: cookie: APISID=027VDklw8fgwAzc9/A5sDm-Sw8SL45HpRB
Header: cookie: SAPISID=P68ohP1FWu9vTYT6/AqWC7LJpg1qJDVChz
Header: cookie: CONSENT=YES+ES.es+V7
```

Figura 4.9: Cabeceras Frame HEADERS

4.1.4.2. Mecanismos de fiabilidad en HTTP/2

En HTTP/1.1 si ocurre cualquier error durante el envío de peticiones no existe ningún mecanismo que permita saber si está ocurriendo un error y por qué. Algunos de esos errores pueden ocurrir antes de que se procese la petición, por ello, en HTTP/2 se han ideado 2 mecanismos para asegurar al cliente que la solicitud no ha sido procesada.

En primer lugar, el *frame* de tipo `GOAWAY` que indica el último flujo que se debería haber procesado. Y en segundo lugar el error de tipo `REFUSED_STREAM` que se incluye en los *frames* `RST_STREAM` y sirve para

indicar que el flujo se va a cerrar antes de procesar ningún *frame*.

Si las solicitudes no son procesadas, evidentemente el receptor del mensaje aún no se ha dado cuenta, por lo que existe la posibilidad de que se puedan reenviar esas solicitudes corrigiendo los errores que se hayan producido.

Además, en este protocolo se incorpora el *frame* de tipo `PING` el cual puede ser muy útil para saber si la conexión entre cliente-servidor funciona correctamente sin necesidad de mandar ninguna solicitud.

4.1.4.3. TLS

TLS es en protocolo a nivel de transporte el cual, mediante la encriptación de los datos que viajan bajo ese protocolo, intenta que las comunicaciones sean seguras.

Todas las implementaciones de HTTP/2 que funcionen bajo conexión segura deberán hacer uso de TLS 1.2 o superiores. Además, la implementación de TLS que se utilice deberá soportar SNI (*Server Name Indication*) [11] ya que cuando HTTP/2 negocia TLS debe hacer referencia al nombre del dominio.

Si se utilizan las versiones TLS 1.3 o superiores en el servidor web, únicamente necesitan soportar SNI sin ningún otro requisito. Sin embargo, si la versión es TLS 1.2 debe cumplir algunos otros requerimientos que en caso de no cumplirse, supondrán que la conexión HTTP/2 finalizará.

Esos requerimientos extra son:

- Se debe desactivar la compresión de datos en TLS.
- La renegociación de TLS debe estar deshabilitada.
- No se deberán incluir ninguno de los conjuntos de cifrados incluidos en el Anexo A de la RFC de HTTP/2.

4.1.4.4. Extensión del protocolo

HTTP/2 permite que el protocolo sea extendido para proporcionar nuevos servicios y/o modificar alguna funcionalidad actual del protocolo.

Algunas extensiones permitidas por el protocolo son:

- Nuevos tipos de *frames*.
- Nuevas opciones para los *frames* de tipo `SETTINGS`.
- Nuevos códigos de error.
- Modificaciones de la semántica del protocolo. Aunque se permite que la semántica del protocolo sea modificada, por ejemplo para cambiar la estructura de los *frames* antes de que ésta modificación pueda utilizarse esta debe ser negociada y aprobada por todos los participantes(clientes y servidores) de la conexión.

4.2. HPACK

4.2.1. Introducción

HPACK está definido en la RFC7541 y es el protocolo que utiliza HTTP/2 para comprimir las cabeceras.

En HTTP/1.1 las cabeceras no utilizaban ningún tipo de compresión. Como ya expliqué en el apartado 4.1.1, cuando se hacen varias peticiones a un servidor web, se están enviando muchas cabeceras duplicadas y que se podrían ahorrar. Debido a que las páginas web son cada vez más complejas y, por tanto, necesitan un mayor número de peticiones por cada página web, se ideó este protocolo que, en combinación con HTTP/2, pretende mejorar la velocidad de carga de las páginas web.

SPDY, antes de que se empezara a idear HTTP/2, utilizaba el formato de compresión DEFLATE [8], el cual eliminaba los campos redundantes de la cabecera, sin embargo, ese formato era vulnerable ante algunos ataques como CRIME [18].

Por ello, junto con la definición de HTTP/2, se ideó este protocolo de compresión que cumple la misma función que DEFLATE pero que añade seguridad al formato de compresión para evitar todos los ataques conocidos en la actualidad.

Este protocolo trata los campos de las cabeceras como si fueran una colección ordenada de nombres y valores, pudiendo tener alguno de estos pares duplicados. Esos pares duplicados, son precisamente los que el protocolo

trata de reducir.

Para trabajar con las cabeceras se crea una tabla codificada en la que cada campo de la cabecera tiene un valor indexado. Estas tablas se pueden actualizar conforme se van recibiendo más cabeceras.

4.2.2. Proceso de codificación

El protocolo HPACK no define un algoritmo de compresión de datos sino que define cómo deben operar los decodificadores de manera que los desarrolladores pueden jugar con la forma de codificar las cabeceras teniendo en cuenta siempre lo que este protocolo permite.

HPACK dice que se debe mantener el orden en el que vienen las cabeceras. Por lo que, tanto al codificar como al decodificar las cabeceras, ese proceso debe realizarse de manera ordenada.

Cada decodificador utiliza una tabla dinámica independiente de las tablas del resto de participantes.

Para codificar las cabeceras son necesarias dos tablas que permitan asociar los campos de la cabecera a unos índices. Para ello se define una tabla estática que contiene los campos de cabecera más comunes y una tabla dinámica que se puede utilizar para indexar los campos de cabecera repetidos.

La tabla estática es una lista de cabeceras predefinidas cuyas entradas se pueden encontrar en el Anexo A de la RFC de HPACK.

La tabla dinámica es una lista dinámica de cabeceras que opera en modo FIFO, es decir, salen los primeros que entran a la tabla. Las entradas más recientes se guardan con los índices más bajos y las entradas más antiguas tienen los índices más altos.

Al inicio de una conexión HTTP/2 esta tabla está vacía y se va rellenando conforme los bloques de cabecera se van descomprimiendo. El codificador tiene libertad para elegir el funcionamiento de esta tabla (memoria destinada, llevar el control de la tabla, decidir cómo actualizarla). Sin embargo, existe un campo de cabecera HTTP/2 (SETTINGS_HEADER_TABLE_SIZE) con el que se puede limitar la memoria destinada a la tabla dinámica.

Por otro lado, ambas tablas están combinadas en un espacio de direcciones común. Desde el índice 1 hasta el índice cuyo valor es la longitud de la tabla estática corresponden a la tabla estática y los índices inmediatamente superiores se refieren a los elementos de la tabla dinámica.

4.2.3. Representación de los campos de la cabecera

Un campo de la cabecera cuando está codificado se puede representar tanto con un índice como con un literal. El índice puede hacer referencia a la tabla estática o a la dinámica y el valor literal se representa con el nombre del campo de la cabecera y su valor.

La representación literal del nombre de un campo de cabecera o del valor de la cabecera puede codificar la secuencia de octetos de forma directa o utilizando codificación Huffman [12] (De forma binaria).

4.2.4. Decodificación del bloque de cabeceras

El decodificador procesa el bloque de cabeceras de forma secuencial ya que asume que las cabeceras vienen ordenadas.

Cuando un campo de cabecera se decodifica y se añade a la lista de cabeceras, ese campo se puede enviar a la aplicación de forma segura. Debido a esto, el decodificador apenas necesita memoria en comparación con la tabla dinámica puesto que el decodificador conforme va decodificando cabeceras las puede ir entregando a la aplicación.

4.2.5. Gestión de la tabla dinámica

Para limitar los requisitos de memoria en el lado del decodificador, la tabla dinámica está limitada en tamaño.

El tamaño de una tabla dinámica se define como la suma de los tamaños de todas sus entradas. Y el tamaño de las entradas es la suma de la longitud de los valores de sus campos en octetos más la longitud del nombre de esa entrada en octetos. A eso se le suman otros 32 octetos que se dejan como seguridad por si en algún caso hiciese falta ese espacio.

Como ya he comentado en ocasiones anteriores, los protocolos que utilizan HPACK pueden determinar el tamaño máximo que tendrá la tabla dinámica. Sin embargo, el codificador no está obligado a utilizar toda esa

capacidad, puede utilizar menos capacidad del tamaño máximo pero, en ningún caso, superarlo.

Además, se permite que durante el transcurso de la conexión HTTP/2 se produzcan actualizaciones del tamaño máximo de la tabla. Esto tiene una utilidad y es que, si se quiere eliminar toda la información de la tabla, se puede poner su tamaño a cero y luego volver a darle un valor distinto de cero al tamaño. De este modo, se habrán borrado todas las entradas de la tabla y se habrá definido un nuevo tamaño para la misma.

Consideraciones a tener en cuenta:

- Si se reduce el tamaño máximo de la tabla, las entradas del final de la tabla dinámica (las primeras que entraron) se van descartando hasta el tamaño de la tabla sea igual al nuevo tamaño máximo definido.
- Si se intenta añadir una nueva entrada a la tabla dinámica y el resultado de esa inserción hace que el tamaño de la tabla sea menor o igual que el tamaño máximo permitido, se insertará esa entrada sin ningún problema ni consecuencia. Si se intenta añadir una entrada que suponga superar el tamaño máximo de la tabla, la tabla se vaciará completamente pero no se tratará como ningún tipo de error.
- Una nueva entrada de la tabla dinámica puede hacer referencia al nombre de una entrada ya existente en esa misma tabla, la cual será eliminada para dar paso a la nueva inserción sin que se supere el tamaño máximo establecido.

4.2.6. Representaciones de tipo primitivo

HPACK utiliza dos tipos primitivos para codificar los datos: números enteros sin signo de longitud variable y cadenas de caracteres.

4.2.6.1. Números enteros

Los números enteros se utilizan para representar los índices de los nombres, los índices de los campos de cabecera o las longitudes de las cadenas de caracteres. La representación de un número entero podrá comenzar en cualquier parte de un octeto, sin embargo, deberá concluir al final del octeto de manera que se facilita el procesamiento de los datos al saber que cuando acaba el octeto, acaba también el número.

El entero se representa dividido en dos partes: un prefijo que contiene un valor numérico y una lista de octetos opcional que se rellenará en caso de que el número no quepa en el octeto para que se se pueda saber con facilidad donde continúa la representación de ese número entero.

4.2.6.2. Cadena de caracteres

Los nombres y los valores de los campos de la cabecera se pueden representar como una cadena de caracteres. Esta cadena puede codificar cada octeto de forma directa o utilizando código Huffman.

La representación de una cadena de caracteres se compone de 3 campos:

- `H`: Es un campo de un solo bit que sirve para indicar si el contenido de la cadena de caracteres está codificado con código Huffman o no.
- `String Length`: Indica el número de octetos utilizados para codificar la cadena de caracteres.
- `String Data`: Va el contenido de la cadena de caracteres, si el campo `H` fuera igual a 1, la cadena de caracteres irá codificada con código Huffman.

Como los datos codificados con código Huffman puede que no acaben al final del octeto, en los casos en los que eso ocurra será necesario añadir *padding* para completar ese octeto. Hay que tener en cuenta que el tamaño de ese *padding* nunca podrá ser superior a 7 bits puesto que eso significaría que hay un *padding* de 1 octeto. Al decodificar la información, el *padding* será descartado.

4.2.7. Seguridad del protocolo

HPACK reduce la longitud de las cabeceras en protocolos como HTTP/1.1 o HTTP/2 eliminando la redundancia de esos paquetes. De esa manera se reduce la cantidad de datos necesarios para enviar tráfico por Internet.

Un atacante podría codificar algunos paquetes y transmitirlos y de esa manera observar la longitud de los paquetes y como varían estos paquetes con la codificación. De esa manera, y tras una serie de pruebas, podría adivinar los datos codificados de los paquetes. Esto es posible a pesar de hacer uso de TLS. El tipo de ataque CRIME se aprovechaba de ese hecho

adivinando un carácter cada vez que mandaba sus paquetes codificados.

HPACK reduce pero no elimina completamente los ataques basados en CRIME [18] ya que obliga al atacante a encontrar una coincidencia de una cabecera completa y no de un solo carácter como les servía con DEFLATE.

Por otro lado, se ha hecho un estudio que confirma que no hay ataques posibles que funcionen contra una codificación Huffman [20] de los datos. Aunque se pudiera robar la información, sería imposible recuperar información útil de esos datos.

Otro ataque que podría probar un atacante sería provocar que se acabara la memoria del punto final de la conexión pero este protocolo tiene el uso de la memoria limitada por el tamaño máximo de la tabla dinámica por lo que tampoco podrían sacar provecho de ese tipo de ataques.

Como última medida de seguridad, el protocolo limita el tamaño de los números enteros y de las cadenas de caracteres.

4.3. Estudios similares

Para la elaboración de este Trabajo de Fin de Grado se ha consultado documentación previamente a iniciar este trabajo. Al inicio de este proyecto no había una RFC del protocolo para consultar sino que sólo se podía consultar un *draft* el cual estaba sujeto a cambios. Además, apenas había implementaciones se servidores o clientes HTTP/2 con las que poder hacer pruebas de rendimiento fiables.

Debido a ello, la mayoría de estudios similares que se encontraron al inicio de esta trabajo eran en base a SPDY y no a HTTP/2.

4.3.1. Estudios similares anteriores basados en SPDY

Como base para realizar mi Trabajo de Fin de Grado me inspiré principalmente en el estudio denominado "*Can SPDY really make the web faster?*" [22]. Este estudio realiza un análisis de rendimiento del protocolo SPDY en dos contextos. Uno en el que se comprueba su eficacia en algunas webs Top100 del ranking Alexa, el cual acaba concluyendo que para las webs de Google se produce un aumento de eficacia de hasta el 20 % en algunos casos, pero ese rendimiento no aumenta en el resto de páginas webs e incluso

llega a empeorar el rendimiento en un 10 % en webs como `Twitter`.

La segunda mitad del estudio analiza el protocolo en un entorno de pruebas controlado en el cual quieren comprobar la eficacia del protocolo ante variaciones en el ancho de banda de la red, pérdida de paquetes y retrasos en la red. En este estudio se concluye que aunque el rendimiento de SPDY mejora a HTTP/1.1 en algunos casos, no es una mejora muy significativa. De hecho, en el artículo se comenta que para entornos con muchas pérdidas el protocolo es mucho menos eficiente que HTTP/1.1.

Existen otros estudios que también analizan la eficacia del protocolo SPDY frente a HTTP/1.1 como: "*A comparison of SPDY and HTTP performance*" [16]. Es un estudio previo al citado anteriormente en el que se deduce que el protocolo no supone una mejora demasiado significativa respecto a su versión anterior.

4.3.2. Estudios similares en la actualidad basados en HTTP/2

En la actualidad existe un estudio llamado "*Is HTTP/2 Really Faster Than HTTP/1.1?*" [7] el cual realiza un análisis del protocolo HTTP/2 muy similar al comentado en el apartado anterior que utilicé para basarme en mi Trabajo de Fin de Grado. Este estudio realiza las mismas pruebas pero en lugar de hacerlas utilizando SPDY ya las realiza sobre HTTP/2. Al igual que el estudio anterior, la conclusión es que la mejora no es demasiado significativa y que en situaciones de altas pérdidas HTTP/2 es menos eficiente que HTTP/1.1.

Otro estudio muy interesante publicado recientemente es "*Power efficient mobile video streaming using HTTP/2 server push*" [21] en el cual se analiza la eficiencia de la nueva característica de HTTP/2 `Server push`. La conclusión de este estudio es que utilizando esa característica, se puede llegar a ahorrar hasta un 17 % de batería en los dispositivos móviles gracias a las peticiones que el teléfono se ahorra enviar ya que el servidor las envía al cliente sin que este las llegue a solicitar.

Capítulo 5

Preparación del entorno de pruebas

5.1. Análisis de los clientes HTTP/2

Actualmente, casi la totalidad de los navegadores web disponibles soportan HTTP/2, pero al inicio de esta investigación solo algunos soportaban esta funcionalidad. En la web `http://caniuse.com/#feat=http2` se puede comprobar qué versión de los principales navegadores web implementó por primera vez este protocolo.

Centrándome en el análisis concreto de qué cliente web utilizar, rápidamente llama la atención que todas las implementaciones existentes de este protocolo solo funcionan sobre TLS. Es decir, no existen navegadores web que funcionen sin cifrar, tan solo existen algunas aplicaciones que realizan peticiones en claro, pero no son navegadores de escritorio. Es una apuesta de los principales navegadores por la estandarización de las conexiones cifradas.

Al inicio de esta investigación, los navegadores web disponibles y sus versiones eran las siguientes:

Cuadro 5.1: Navegadores compatibles con HTTP/2

Navegador	**Versión**	**Sistema Operativo**
Google Chrome	44.0.2403.39	Windows
Google Chrome	43.0.2357.93	Andorid/IOS
Chromium	43.0.2357.93	Linux
Firefox	38.0.5	Windows/Linux
Opera	30.0.1835.59	Windows/Linux
Internet Explorer	11	Windows 10

De todos estos clientes se decidió hacer uso de los navegadores web de Google. La versión windows para realizar las capturas de paquetes debido a que no existía una versión de wireshark en linux que soportara HTTP/2, y la versión linux para utilizar el script programado de automatización de pruebas.

En la actualidad, hay más navegadores que ofrecen soporte para HTTP/2 como Safari, Microsoft Edge, Firefox para móviles, Opera para móviles y la nueva versión de Internet Explorer 11 compatible con Windows 7 y Windows 8.1.

5.2. Captura de paquetes HTTP/2

Para capturar los paquetes HTTP/2 hay dos maneras de hacerlo: una primera desde el propio navegador de Google y otra haciendo uso de otros programas como Wireshark. A priori, la ventaja que ofrece Wireshark es que permite realizar capturas utilizando diferentes navegadores web por lo que también permite analizar como gestiona cada navegador el nuevo protocolo.

5.2.1. Capturas desde el navegador Web

Este tipo de capturas sólo se pueden hacer desde el navegador de Google, es por eso por lo que decidí hacer uso de este navegador desde el principio de la investigación.

Para realizar dicha captura, se debe escribir lo siguiente en el navegador:

`chrome://net-internals/`

Cuando se abre la ventana, en el cuadro desplegable hay que seleccionar la opción HTTP/2 para que se muestren todas las sesiones activas en el navegador.(Ver imagenes 5.1 y 5.2).

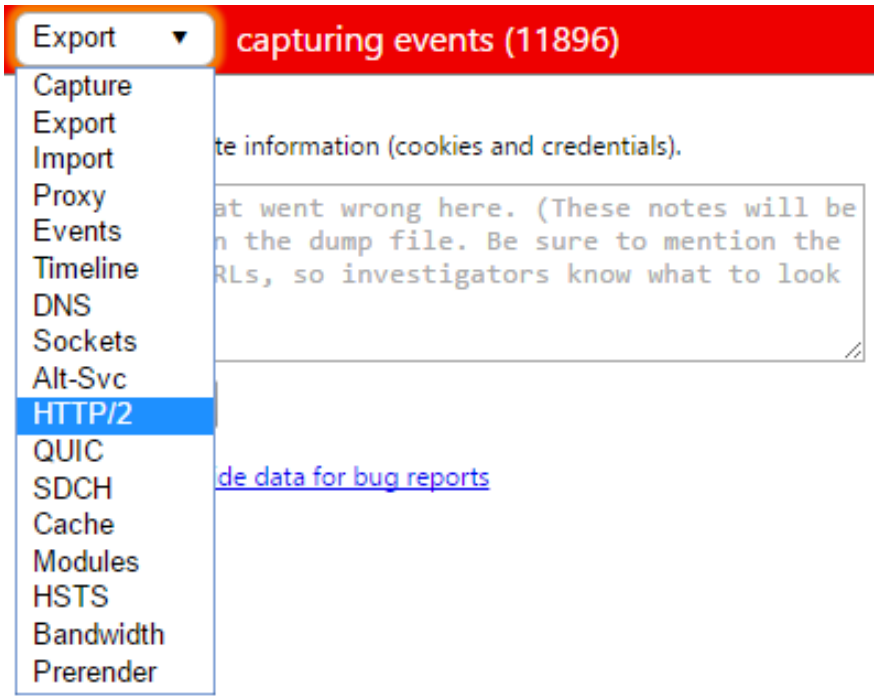


Figura 5.1: Captura de paquetes con Google Chrome-1

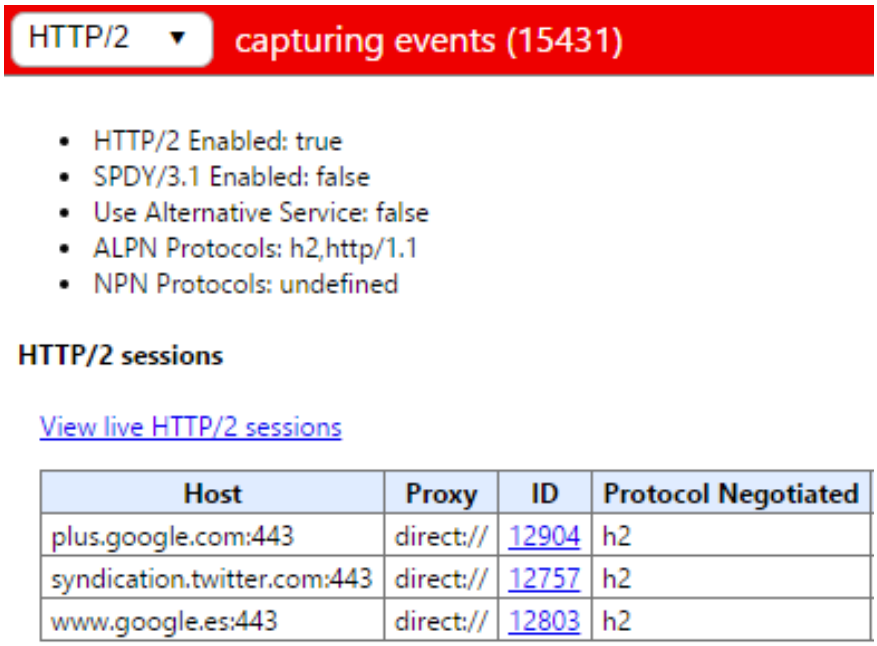


Figura 5.2: Captura de paquetes con Google Chrome-2

A continuación, si se hace click sobre el id de la sesión, se abrirá una nueva ventana en la que se puede observar el intercambio de *frames* de dicha sesión. (Ver imagen 5.3).

```
   12803: HTTP2_SESSION
Start Time: 2016-06-18 18:26:38.763
t= 152 [st=   0]  HTTP2_SESSION_SEND_HEADERS
                  --> exclusive = true
                  --> fin = true
                  --> has_priority = true
                  --> :method: GET
                      :authority: www.google.es
                      :scheme: https
                      :path: /gen_204?v=3&s=newtab&atyp=csi
&p=s&npn=1&ima=1&rt=prt.5,xjsls.5,ol.8,xjses.31,
                      accept: image/webp,image/*,*/*;q=0.8
                      user-agent: Mozilla/5.0 (Windows NT 6.3; WOW64)
AppleWebKit/537.36 (KHTML, like Gecko) Chrome/51.0.2704.103 Safari/537.36
                      x-client-data: CIu2yQEIo7bJAQ==
                      referer: https://www.google.es/_/chrome/newtab?espv=2&
ie=UTF-8
                      accept-encoding: gzip, deflate, sdch, br
                      accept-language: es-ES,es;q=0.8
                      cookie: [699 bytes were stripped]
                  --> priority = 2
                  --> stream_id = 69
t= 180 [st=  28]  HTTP2_SESSION_RECV_HEADERS
                  --> fin = false
                  --> :status: 200
                      version: 124817099
                      content-type: application/json; charset=UTF-8
                      content-disposition: attachment; filename="f.txt"
                      content-encoding: gzip
                      date: Sat, 18 Jun 2016 16:26:32 GMT
                      server: gws
                      content-length: 56085
                  --> stream_id = 69
```

Figura 5.3: Captura de paquetes con Google Chrome-3

5.2.2. Capturas utilizando Wireshark

El uso de programas como Wireshark o tcpdump para la captura de paquetes HTTP/2 es algo más complejo debido a que el protocolo HTTP/2 se utiliza sobre conexiones cifradas. Esto implica que no se puede realizar una captura de tráfico y ver el contenido de los paquetes HTTP/2. Para poder ver ese contenido, es necesario descifrar los paquetes para lo que hará falta averiguar la clave privada del cifrado en primer lugar y, a continuación, habrá que decirle a wireshark cuál es esa clave.

5.2.2.1. Descifrado de paquetes en wireshark

En primer lugar, es necesario obtener la clave privada con la cual se han encriptado los paquetes HTTP/2. Para ello, se establece una variable de entorno llamada `SSLKEYLOGFILE`. En esa variable de entorno se indica la ruta en la cual se encuentra el fichero sobre el que se van a ver reflejadas las claves privadas utilizadas para las conexiones cifradas. Esta variable de entorno solo funciona con los navegadores Google Chrome y Firefox.

Una vez ya se ha obtenido la clave privada, sólo queda indicar a wireshark cuál es esa clave para que se puedan capturar paquetes y ver su contenido. Para ello, es necesario abrir el menú "Edit - Preferences". A continuación, aparecerá una nueva ventana de configuraciones en la que hay que seleccionar el desplegable "Protocols".

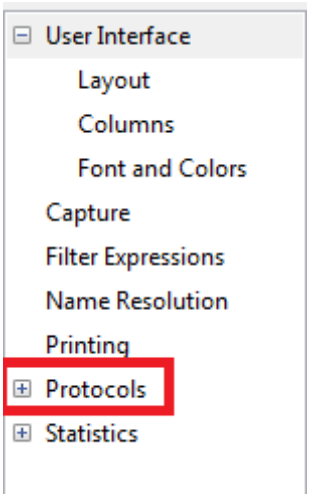


Figura 5.4: Configuración de wireshark-1

Cuando se abra el desplegable, se selecciona el protocolo SSL y en el

campo “(Pre)-Master-Secret log filename” se incluirá el archivo que se creó anteriormente al definir la variable de entorno.

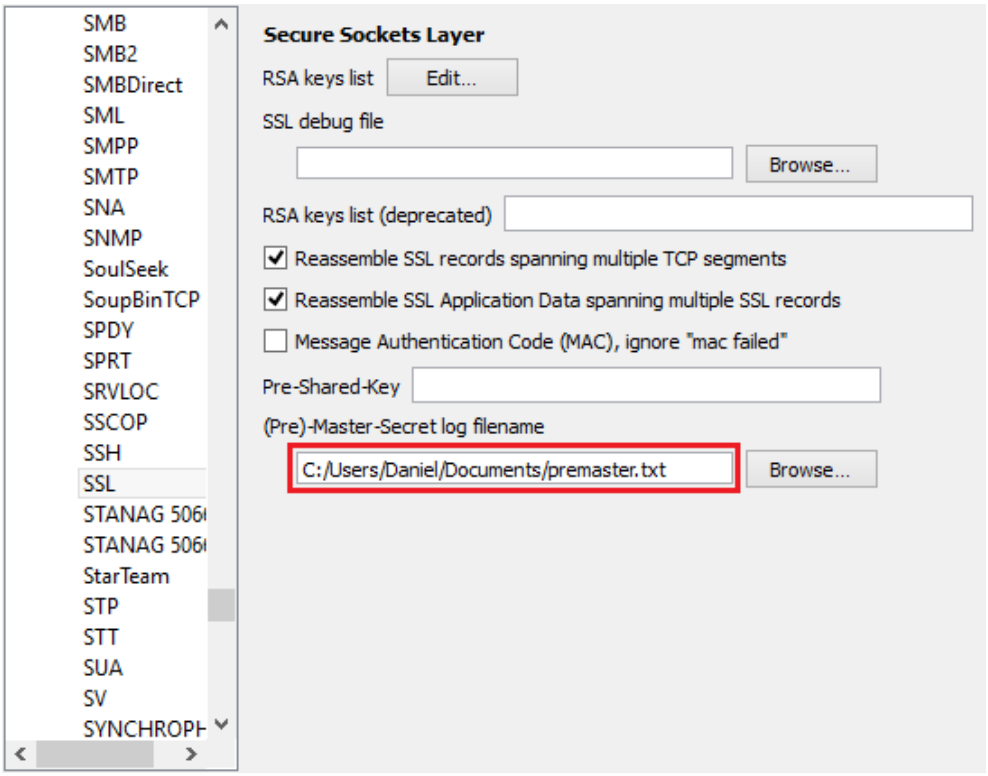


Figura 5.5: Configuración de wireshark-2

Ya sólo queda configurar wireshark para que muestre los paquetes HTTTP/2 ya que por defecto interpreta que el formato utilizado es el del protocolo SPDY (este paso ya no es necesario en versiones más recientes de wireshark debido a que soportan HTTP/2 por defecto).

Para ello, al igual que se ha seleccionado anteriormente el protocolo SSL se buscaría el protocolo HTTP y cuando se abren las opciones disponibles sólo hay que activar el check que aparece en pantalla.

Si ahora se realiza una captura de paquetes de cualquier página web cuyo servidor soporte HTTP/2, se podrán observar las tramas como si no estuvieran cifradas.

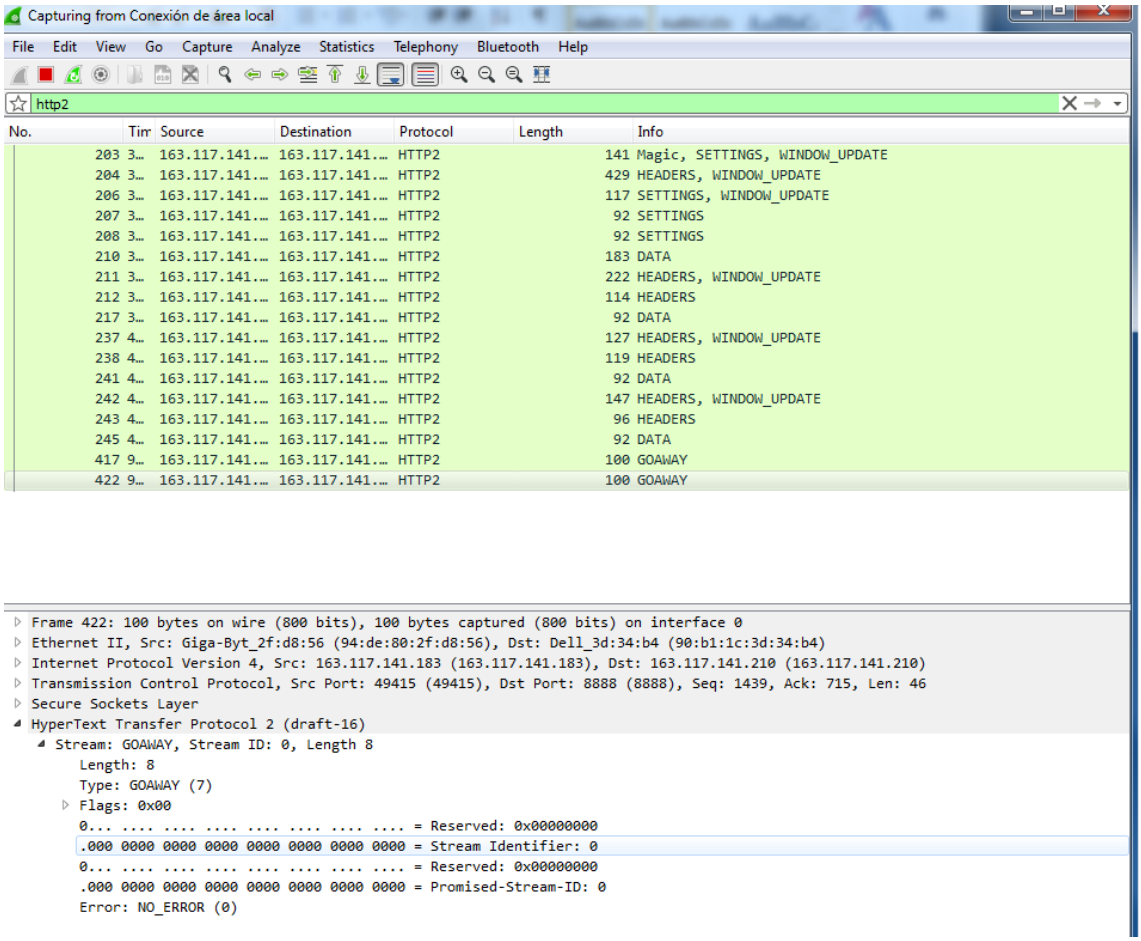


Figura 5.6: Ejemplo de captura

5.3. Análisis de los servidores HTTP/2

Al inicio de esta investigación se realizó un estudio de los distintos servidores disponibles que implementaran tanto HTTP/2 como HPACK. Al igual que ocurre con los navegadores web, los servidores HTTP/2 obligaban a hacer uso de conexiones cifradas. Actualmente, algunos servidores sí soportan HTTP/2 sin conexiones cifradas pero no es lo más habitual.

Los servidores disponibles al inicio de mi investigación que soportaran HTTP/2 y HPACK eran los siguientes:

Cuadro 5.2: Servidores compatibles con HTTP/2

Servidor	**Sistema Operativo**
IIS	Windows
Open Lite Speed	Linux
Apache	Linux
H2O	Linux
nghttp2	Linux

Después de realizar algunas pruebas con los servidores, se decidió continuar con la etapa de pruebas utilizando el servidor "Open Lite Speed" ya que era el que más avanzada tenía la implementación del protocolo y el que más facilidades proporcionaba. Por ejemplo, se podía lanzar en un puerto un servidor HTTP/2 y en otro puerto un servidor HTTP/1.1 sobre TLS(HTTPs) por lo que además, era el servidor más eficiente con los recursos de los que disponía.

5.4. Puesta a punto del servidor Open Lite Speed

Para realizar la instalación, se solicitó al departamento de Telemática de la Universidad Carlos III de Madrid un servidor con sistema operativo Linux sobre el que poder realizar la instalación.
Nos proporcionaron un servidor Ubuntu 12.04.5 LTS denominado **Halffter** sobre el cual se realizó la instalación de Open Lite Speed versión 1.3.9.

Para poder utilizar el servidor con HTTP/2 había que realizar una serie de configuraciones que detallo a continuación:

En primer lugar, hay que iniciar el servidor con el comando *sudo /usr/local/lsws/bin/lswsctrl start* ya que tras terminar la instalación del mismo este no inicia automáticamente.

A continuación, hay que abrir el panel de configuración desde la URL `https://halffter.gast.it.uc3m.es:7080` y pulsar sobre la opción "Configuration".

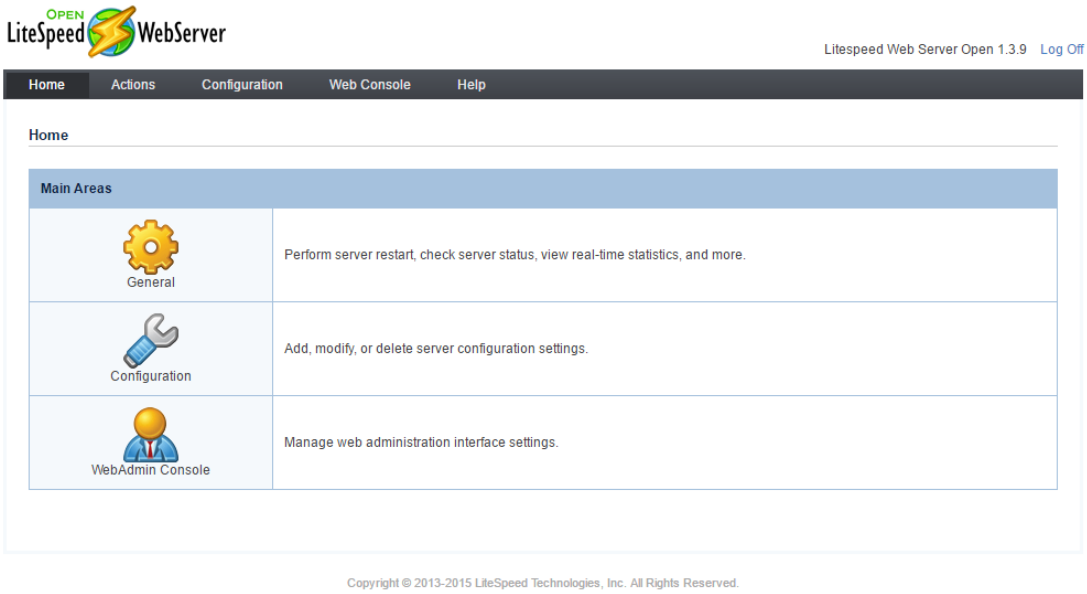


Figura 5.7: Configuración del servidor-1

Tras esto, en la nueva pantalla se seleccionará el menú "Listeners" desde el cual se pueden configurar los protocolos que van a funcionar en cada puerto.

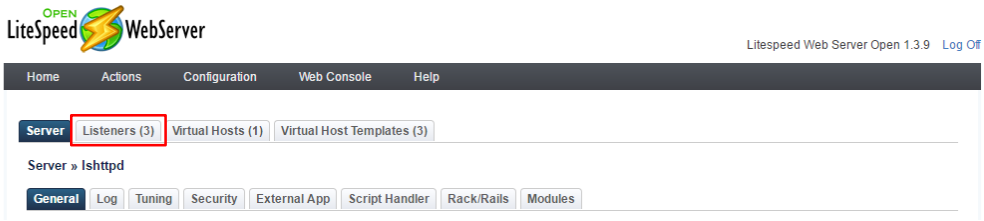


Figura 5.8: Configuración del servidor-2

En esa ventana se pulsaría sobre el botón *add* y se añadiría el nuevo *Listener*. En las imágenes 5.9 y 5.10 se puede observar como queda configurado el servidor para utilizar el protocolo HTTP/2 en el puerto 8888.

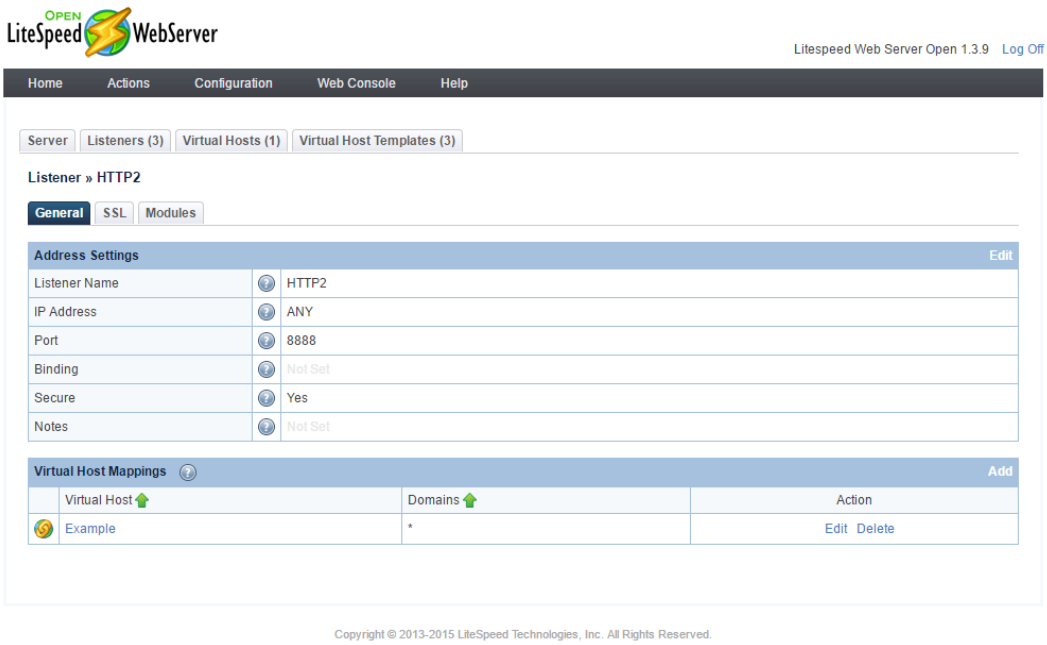


Figura 5.9: Configuración del servidor-3

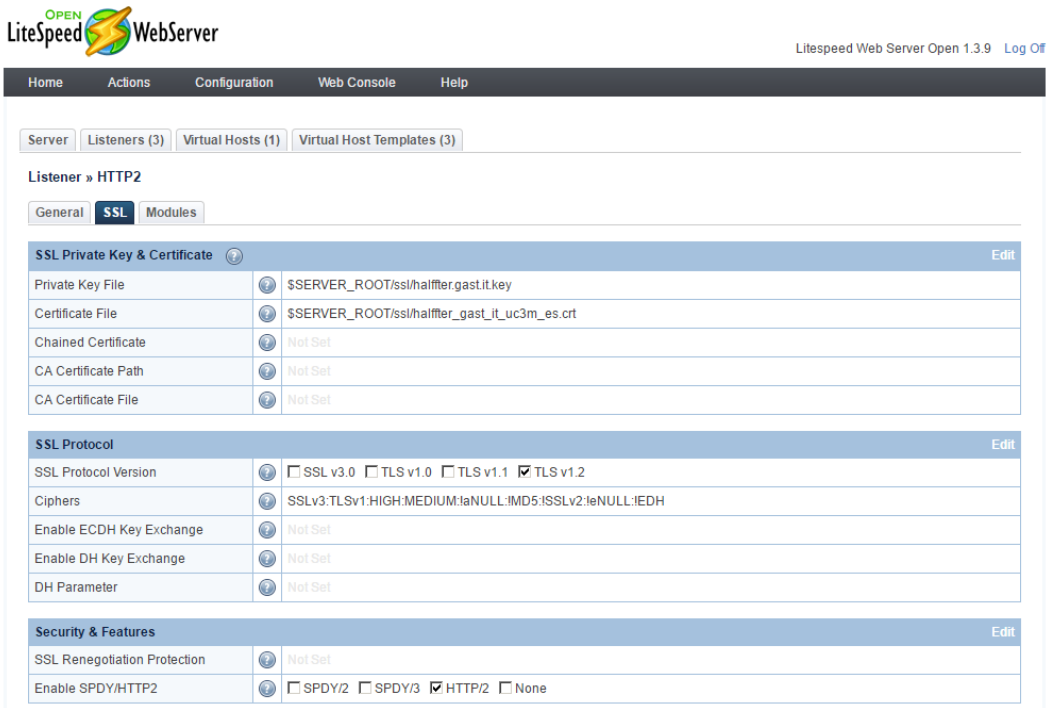


Figura 5.10: Configuración del servidor-4

5.4.1. Preparación de Wireshark para la captura de paquetes del servidor

En la sección 5.2.2.1 preparamos la herramienta para poder capturar paquetes HTTP/2 pero, a pesar de ello aún no se pueden descifrar los que recibamos desde nuestro servidor.

Wireshark configura por defecto una serie de puertos por los que sabe que se están recibiendo paquetes HTTP y HTTPs pero, el servidor que hemos configurado, opera con HTTP/2 en el puerto 8888 por lo que Wireshark, al no tener ese puerto configurado, no interpreta que esos paquetes haya que descifrarlos como si fueran paquetes HTTP. El motivo por el que he escogido otros puertos diferentes a los que vienen por defecto es que la máquina *Halffter* ya tenía instalado otro servidor web en los puertos por defecto. Además, a la hora de simular el estado de la red, se podía configurar para que la pérdida de paquetes, el retraso en la red, o el ancho de banda afectase solo al tráfico dirigido a esos puertos de manera que no se penalizaba el rendimiento de toda la máquina.

Para que Wireshark reconozca esos puertos como tráfico HTTP, hay que ir al menú "Edit - Preferences". Y en la ventana que se abre se selecciona la opción "Protocols" en la que habrá que buscar el protocolo HTTP. Una vez en ese menú de configuración, tan solo habrá que añadir los puertos 8888 y 8887 sobre los que el servidor servirá los protocolos HTTP/2 y HTTPs respectivamente.

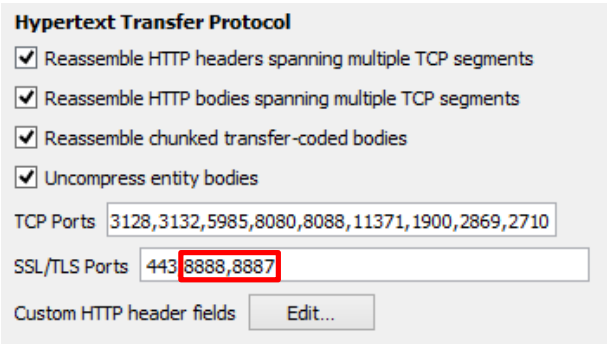


Figura 5.11: Configuración de la captura de paquetes del servidor

Capítulo 6

Metodología seguida para la realización de las pruebas

Para realizar las pruebas de rendimiento, estas se han dividido en 4 fases para evaluar la eficacia de HTTP/2 en diferentes situaciones:

- Añadir un retraso a la red.
- Simular pérdida de paquetes en la red.
- Variar el ancho de banda de la red.
- Medir el tamaño de los paquetes recibidos usando HTTP/2 y HTTP/1.1 sobre conexiones cifradas(HTTPS).

Para ello, se han creado 3 páginas web de distinto tamaño (48 KB, 578 KB y 1.21 MB) y cada una de ellas se ha subdividido en otras 3 con diferente número de recursos(1, 16 y 64). Por lo tanto, cada una de las pruebas anteriores se realizará sobre un total de 9 páginas diferentes. Las simulaciones realizadas para llevar a cabo el análisis de rendimiento se pueden comprobar en la figura 6.1.

A continuación, se detallará todo el proceso que se necesita para ejecutar las pruebas de rendimiento, desde como se crean las diferentes páginas web hasta la elaboración de un script que automatiza toda la realización de estas pruebas.

6.1. Creación de los archivos html para las pruebas

Como esta versión de HTTP ha sido ideada para mejorar la eficiencia del protocolo reduciendo el tiempo de carga y la latencia, las pruebas se realizarán utilizando páginas web de diversos tamaños y diferente número de recursos para comprobar como se comporta el protocolo en distintas situaciones.

Como se ha comentado en la introducción de este capítulo, se han diseñado 3 páginas web de diferentes tamaños. Para que el tamaño de la página fuera diferente, estas estaban formadas por diferentes imágenes en formato BMP. En este caso, el formato de las imágenes en este caso era importante puesto que cada imagen iba a dividirse en 16 y 64 trozos además de la imagen original y la manera más eficiente de que la imagen original ocupara el mismo tamaño que la suma de los trozos en los que se había dividido, era que la imagen estuviera en un formato sin compresión.

Aquí se puede ver un ejemplo de una web que utiliza una sola imagen o esa misma web construida con la imagen dividida en 16 fragmentos.

```
<html>
<head><title>Imagen 1x1</title></head>
<body>
<img src="paisaje.bmp"/>
</body>
</html>
```

index1.html

```
<html>
<head><title>Imagen 4x4</title></head>
<body>
<table border="0" cellpadding="0" cellspacing="0">
<tr>
 <td><img src="paisaje (1).bmp" alt="Part 1x1" border="0" /></td>
 <td><img src="paisaje (2).bmp" alt="Part 1x2" border="0" /></td>
 <td><img src="paisaje (3).bmp" alt="Part 1x3" border="0" /></td>
 <td><img src="paisaje (4).bmp" alt="Part 1x4" border="0" /></td>
</tr>
<tr>
 <td><img src="paisaje (5).bmp" alt="Part 2x1" border="0" /></td>
 <td><img src="paisaje (6).bmp" alt="Part 2x2" border="0" /></td>
 <td><img src="paisaje (7).bmp" alt="Part 2x3" border="0" /></td>
 <td><img src="paisaje (8).bmp" alt="Part 2x4" border="0" /></td>
</tr>
<tr>
 <td><img src="paisaje (9).bmp" alt="Part 3x1" border="0" /></td>
 <td><img src="paisaje (10).bmp" alt="Part 3x2" border="0" /></td>
 <td><img src="paisaje (11).bmp" alt="Part 3x3" border="0" /></td>
 <td><img src="paisaje (12).bmp" alt="Part 3x4" border="0" /></td>
```

```
</tr>
<tr>
 <td><img src="paisaje (13).bmp" alt="Part 4x1" border="0" /></td>
 <td><img src="paisaje (14).bmp" alt="Part 4x2" border="0" /></td>
 <td><img src="paisaje (15).bmp" alt="Part 4x3" border="0" /></td>
 <td><img src="paisaje (16).bmp" alt="Part 4x4" border="0" /></td>
</tr>
</table>
</body>
</html>
```

index16.html

6.2. Simulación del estado de la red

Para poder simular el estado de la red, se configuraron un cliente y un servidor además de un nodo intermedio que serviría para realizar las simulaciones de retrasos, pérdidas de paquetes y cambios en el ancho de banda.

En la figura 6.1 se puede comprobar el esquema del entorno de red utilizado para las pruebas.

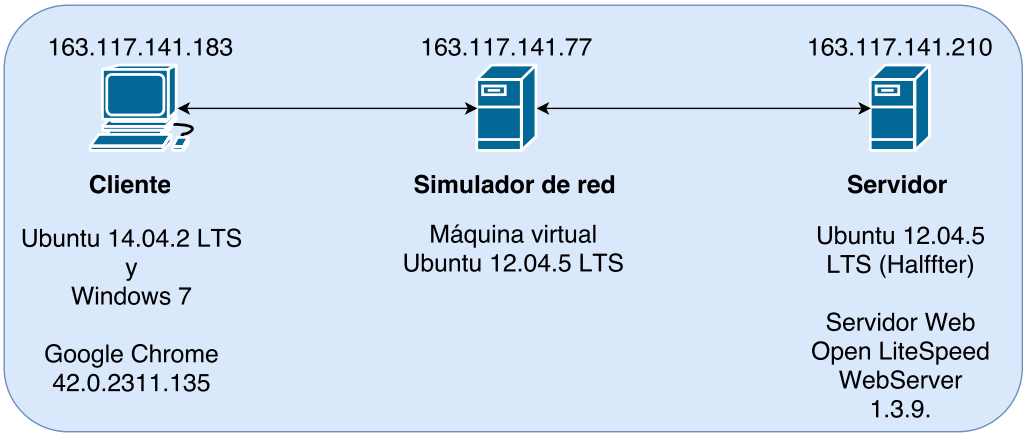


Figura 6.1: Entorno de red

6.2.1. Configuración del cliente

En el lado del cliente hubo que hacer dos configuraciones diferentes, una en el cliente con Linux que obligara a que los paquetes dirigidos al servidor pasaran por el nodo intermedio y otra para que el cliente con Windows hiciera lo mismo.

En Windows hubo que crear una ruta estática que obligara a los paquetes dirigidos al servidor a pasar por el nodo intermedio. Para configurar la ruta estática se utilizó el siguiente comando:

```
route -p add 163.117.141.210 mask 255.255.255.255 163.117.141.77
```

Al igual que en Windows, en Linux también se tuvo que crear una ruta estática. En Linux el comando utilizado fue el siguiente:

```
Sudo ip route add 163.117.141.210/32 via 163.117.144.77 dev p4p1
```

6.2.2. Configuración del servidor

En el servidor también hubo que configurar una ruta estática de manera que todas las respuestas HTTP que fueran hacia el cliente también pasaran por ese nodo intermedio en el que se iban a simular las condiciones de la red.

La configuración en el servidor se hizo utilizando el siguiente comando:

```
Sudo ip route add 163.117.141.183/32 via 163.117.144.77 dev eth0
```

6.2.3. Configuración del nodo intermedio

En el nodo intermedio lo primero que hubo que hacer es configurarlo para que se comportara como un router de manera que encaminara el tráfico entre cliente y servidor. Para ello hubo que entrar en el archivo */etc/sysctl.conf* y modificar la opción `net.ipv4.ip_forward` poniendo su valor a "1".

Una vez realizada esa configuración en el nodo intermedio ya se podía realizar la simulación de la red y para ello se utilizaron los programas TC [10] y Netem [9].

Con el comando `TC` se puede establecer el tipo de encolado que utilizará nuestra red. En mi caso decidí utilizar una red HTB (*Hierarchical Token Bucket*) ya que me permitía crear clases para afectar solo a los puertos 8888 y 8887 del tráfico que circulara por esa máquina. Es decir, no era necesario utilizar una máquina exclusiva para realizar esta simulación de la red, sino que mientras yo simulaba el tráfico con destino servidor, otros investigadores podrían utilizarla para simular su red con otra configuración diferente sin que nos molestáramos. Además, con este comando se puede simular el ancho de banda utilizado por la máquina para esos puertos, y, en combinación con

`netem`, se puede establecer un ratio de pérdida de paquetes, un retraso en la red, un ratio de paquetes corruptos,etc.

La siguiente tabla muestra todas las configuraciones de red que se han utilizado para realizar las pruebas de rendimiento:

Cuadro 6.1: Configuraciones de red

Condiciones ideales
Pérdidas del 5 %
Pérdidas del 10 %
Pérdidas del 15 %
Pérdidas del 20 %
Retraso de 50 ms
Retraso de 100 ms
Retraso de 200 ms
Retraso de 500 ms
10 Mbps
1 Mbps
500 Kbps
100 Kbps

Cada una de las páginas webs diseñadas en la sección 6.1 fue probada con todas las configuraciones de red citadas en la tabla 6.1.

A continuación adjunto algunos ejemplos comentados de las distintas configuraciones de la tabla 6.1.

```
1 #Se establece que el encolado será de tipo HTB.
2 sudo tc qdisc add dev eth0 root handle 1: htb default 10
3 #Se define el ancho de banda que tiene la máquina. 1 Gbps en este caso.
4 sudo tc class add dev eth0 parent 1: classid 1:1 htb rate 1gbit ceil 1gbit
5 #Se define la clase sobre la que se va a simular el estado de la red. La 20
      en este caso.
6 sudo tc class add dev eth0 parent 1:1 classid 1:20 htb rate 1gbit ceil 1gbit
7 #Se especifican las pérdidas para la clase 20. Un 5% en este ejemplo.
8 sudo tc qdisc add dev eth0 parent 1:20 handle 20: netem loss 5%
9 #Se aplica el filtro a la clase anterior indicando el puerto al que aplicará
       esta configuración de red. En este caso aplicará tanto al puerto 8888(
      HTTP/2) como al 8887(HTTP/1.1 sobre TLS)
10 sudo tc filter add dev eth0 parent 1: protocol ip prio 1 u32 match ip sport
      8888 0xfff flowid 1:20
11 sudo tc filter add dev eth0 parent 1: protocol ip prio 1 u32 match ip sport
      8887 0xff flowid 1:20
```

Pérdidas del 5 %

```
#Se establece que el encolado será de tipo HTB.
sudo tc qdisc add dev eth0 root handle 1: htb default 10
#Se define el ancho de banda que tiene la máquina. 1 Gbps en este caso.
sudo tc class add dev eth0 parent 1: classid 1:1 htb rate 1gbit ceil 1gbit
#Se define la clase sobre la que se va a simular el estado de la red. La 20 en este caso.
sudo tc class add dev eth0 parent 1:1 classid 1:20 htb rate 1gbit ceil 1gbit
#Se especifica el retraso para la clase 20. 200 milisegundos en este ejemplo
sudo tc qdisc add dev eth0 parent 1:20 handle 20: netem delay 200ms
#Se aplica el filtro a la clase anterior indicando el puerto al que aplicará esta configuración de red. En este caso aplicará tanto al puerto 8888(HTTP/2) como al 8887(HTTP/1.1 sobre TLS)
sudo tc filter add dev eth0 parent 1: protocol ip prio 1 u32 match ip sport 8888 0xfff flowid 1:20
sudo tc filter add dev eth0 parent 1: protocol ip prio 1 u32 match ip sport 8887 0xff flowid 1:20
```

Retrasos de 200 ms

```
#Se establece que el encolado será de tipo HTB.
sudo tc qdisc add dev eth0 root handle 1: htb default 10\r
#Se define el ancho de banda que tiene la máquina. 1 Gbps en este caso.
sudo tc class add dev eth0 parent 1: classid 1:1 htb rate 1gbit ceil 1gbit
#Se define el ancho de banda que se va a simular para la clase 20. 500 Kbps en este caso.
sudo tc class add dev eth0 parent 1:1 classid 1:20 htb rate 500kbit ceil 500 kbit
#Se aplica el filtro a la clase anterior indicando el puerto al que aplicará esta configuración de red. En este caso aplicará tanto al puerto 8888(HTTP/2) como al 8887(HTTP/1.1 sobre TLS)
sudo tc filter add dev eth0 parent 1: protocol ip prio 1 u32 match ip sport 8888 0xfff flowid 1:20
sudo tc filter add dev eth0 parent 1: protocol ip prio 1 u32 match ip sport 8887 0xff flowid 1:20
```

Ancho de banda de 500 Kb

6.3. Elaboración del script automático de pruebas

El script se ha realizado utilizando `script bash`, `node.js` y un paquete denominado `Chrome-Har-Capturer` [6] que permitía generar un archivo `.har` con los datos de las pruebas de rendimiento realizadas. Un fichero HAR (*HTTP Archive*) es aquel cuyo contenido está en formato `json` y que sirve para monitorizar la interacción de los navegadores web con las páginas web.

Para la realización de las pruebas de rendimiento, es necesario abrir Google Chrome en modo *benchmarking*, el cual permite realizar pruebas de

rendimiento. Por lo tanto, el primer paso que hay que añadir al script de pruebas es abrir el navegador en este modo de manera automática.

Una vez ya se ha abierto el navegador, se puede comenzar con la ejecución de las pruebas. En primer lugar, el script copia la página html sobre la cual se van a realizar las pruebas y la coloca en la carpeta del servidor. Para ello utiliza este código:

```
#!/usr/bin/expect

spawn ssh davega@halffter.gast.it.uc3m.es
expect "*assword:*"
send "19684dvj\r"
expect "*davega*"
send " sudo tc qdisc del dev eth0 root\r"
expect "*davega:*"
send "19684dvj\r"
expect "*davega*"
send "cd ../../usr/local/lsws/http2/html\r
rm paisaje* index.html\r
cd ..\r
cp 1.21MB/img/* html\r
exit\r"
interact
```

Script para obtener la página sobre la que se realizarán las pruebas

El código citado anteriormente accede al servidor web y, en primer lugar, se asegura de que no haya ninguna simulación de red activa para que el envío de los comandos no sea demasiado lento. Tras esto, accede a la ruta donde se encuentra el html que el servidor está utilizando actualmente, lo elimina y mueve el nuevo archivo html al servidor. A continuación cierra la conexión ssh y el script pasa a la siguiente fase.

La siguiente fase sería simular el estado de la red en el nodo intermedio. Para ello se utiliza el siguiente script:

```
#!/usr/bin/expect

spawn ssh davega@halffter.gast.it.uc3m.es
expect "*assword:*"
send "19684dvj\r"
expect "*davega*"
send " sudo tc qdisc del dev eth0 root\r"
expect "*davega:*"
send "19684dvj\r"
expect "*davega*"
send "sudo tc qdisc add dev eth0 root handle 1: htb default 10\r
sudo tc class add dev eth0 parent 1: classid 1:1 htb rate 1gbit ceil 1gbit\r
sudo tc class add dev eth0 parent 1:1 classid 1:20 htb rate 1gbit ceil 1gbit
    \r
sudo tc qdisc add dev eth0 parent 1:20 handle 20: netem delay 200ms\r
```

```
15 sudo tc filter add dev eth0 parent 1: protocol ip prio 1 u32 match ip sport
       8888 0xfff flowid 1:20\r
16 sudo tc filter add dev eth0 parent 1: protocol ip prio 1 u32 match ip sport
       8887 0xff flowid 1:20\r
17 exit\r"
18 interact
```

Script que configura la conexión de red

Este script accede al nodo intermedio y lo primero que hace es eliminar la simulación de red activa para que el envío de los siguientes comandos sea más rápido. Tras esto, utiliza los programas `TC` y `Netem` para simular las nuevas condiciones de red. En el código anterior se simula un retraso de 200ms.

Por último ya sólo queda realizar las peticiones al servidor web con el cliente en modo *benchmarking*. Para ello se utiliza el paquete Chrome-Har-Capturer que, además de realizar las consultas al servidor, almacenará en un fichero .har el tiempo empleado para realizar dicha captura. A continuación muestro uno de los scripts realizados en `node.js` utilizando dicho paquete:

```
1  var fs = require('fs');
2  var chc = require('chrome-har-capturer');
3  var c = chc.load(['https://halffter.gast.it.uc3m.es:8888/',
4                    'https://halffter.gast.it.uc3m.es:8888/',
5                    'https://halffter.gast.it.uc3m.es:8888/',
6                    'https://halffter.gast.it.uc3m.es:8888/',
7              'https://halffter.gast.it.uc3m.es:8888/',
8              'https://halffter.gast.it.uc3m.es:8888/',
9              'https://halffter.gast.it.uc3m.es:8888/',
10           'https://halffter.gast.it.uc3m.es:8888/',
11           'https://halffter.gast.it.uc3m.es:8888/',
12           'https://halffter.gast.it.uc3m.es:8888/']);
13 c.on('connect', function () {
14     console.log('Connected to Chrome');
15 });
16 c.on('end', function (har) {
17     fs.writeFileSync('out200mshttp2.har', JSON.stringify(har));
18 });
19 c.on('error', function () {
20     console.error('Cannot connect to Chrome');
21 });
```

Script que realiza las peticiones a la página web

En el ejemplo anterior, se están realizando 10 peticiones HTTP/2 ya que se lanzan contra el puerto 8888 y, el resultado de esas 10 peticiones, se almacenará en un fichero llamado "`out200mshttp2.har`".

Para visualizar en forma de gráfico el contenido de los ficheros har se ha utilizado la herramienta `http://www.softwareishard.com/har/viewer/`

Capítulo 7

Resultados

Los siguientes resultados se han obtenido utilizando el procedimiento explicado en el capítulo anterior.

Para comprobar la eficiencia del protocolo y descartar valores puntuales que fueran erróneos se han tomado 100 valores por cada prueba y los resultados obtenidos que se van a comentar a continuación son la media de todos esos valores.

El tiempo utilizado en las mediciones tiene en cuenta desde el primer paquete enviado hasta el último paquete recibido. En estos tiempos se incluye el tiempo de descifrar la conexión TLS, el de espera y el de descarga de la web.

7.1. Retrasos en la red

7.1.1. Resultados esperados

En este caso no hay ninguna duda de que HTTP/2 debería verse muy beneficiado. HTTP/2 debería ganar tiempo respecto a su versión anterior por tres razones:

1. HTTP/1.1 debe establecer múltiples conexiones, por ejemplo, Google Chrome establece 6 conexiones simultaneas. Estas múltiples conexiones implican también un mayor tiempo de negociación para SSL ya que no sólo se negocia una conexión segura sino que HTTPS debe negociar una por cada conexión.

2. La segunda razón es la entrega ordenada de paquetes que exige HTTP/1.1. Mientras que en la versión anterior del protocolo para mandar

una nueva petición había que haber recibido todas las respuestas en el orden correcto, en HTTP/2 se pueden realizar todas las peticiones necesarias, siempre que la ventana de TCP lo permita, sin esperar a recibir una respuesta. Es decir, si en HTTP/2 llega un paquete desordenado o se produce algún retraso en la entrega no pasa nada, se pueden seguir enviando peticiones mientras que la ventana de datos lo permita.

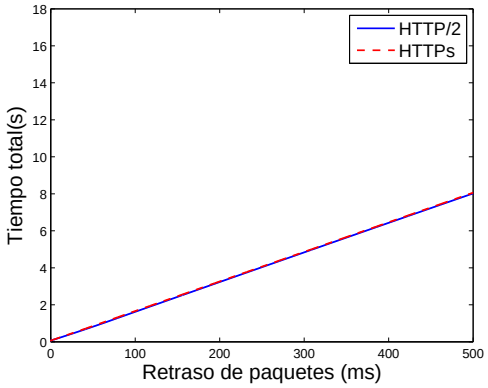
3. En HTTPS, como los navegadores web no tienen activado el *pipelining* por defecto, se debe de esperar a recibir una respuesta por petición, por lo que para cada petición se producirá un retraso a la ida otro a la vuelta. Si se supone un retraso de 500ms en la red, al final se tardaría 1 segundo en enviar la siguiente petición. Sin embargo, en HTTP/2 sí se hace uso de *pipelining* por lo que teniendo el mismo retraso en la red, al poder enviar varias peticiones de manera simultánea, se ahorra ese tiempo de espera en recibir cada respuesta. Aunque HTTPS también puede enviar varias peticiones simultaneas (1 por cada conexión), el protocolo se ve limitado por el número de conexiones establecidas

7.1.2. Resultados obtenidos

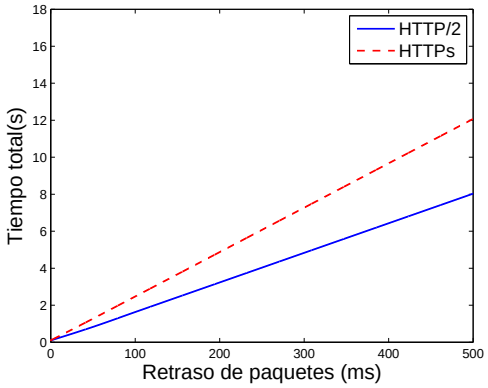
En los gráficos 7.1, 7.2 y 7.3 se observa el comportamiento que tienen tanto HTTP/2 como HTTPS conforme el retraso en la red va aumentando. Para verlo con más claridad, se han divido los resultados en varias imágenes de distintos tamaños y recursos en las cuales se comprueba como afecta el tamaño y número de recursos de las páginas en el rendimiento del protocolo.

Observando las imágenes citadas anteriormente, se puede comprobar como a medida que aumenta el número de paquetes, HTTP/2 mejora el rendimiento respecto a HTTPS. Además, en la imagen 7.3 se observa que incluso cuando la imagen se divide en 1 solo fragmento HTTP/2 mejora respecto a HTTPS, esto se debe a que la página web es mucho mayor que el tamaño de ventana TCP y, por tanto, la imagen está viajando por la red fragmentada.

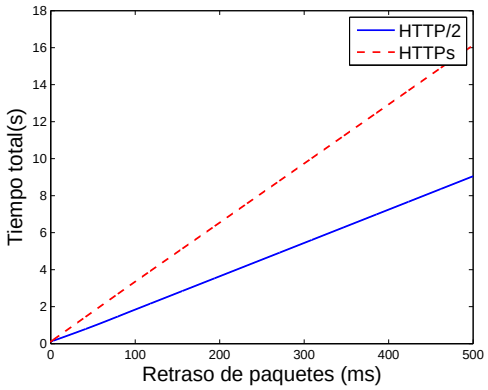
7.1.2.1. Imagen de 48KB



(a) Una división



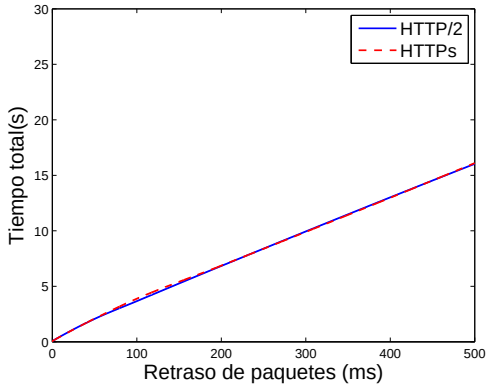
(b) Diez y seis divisiones



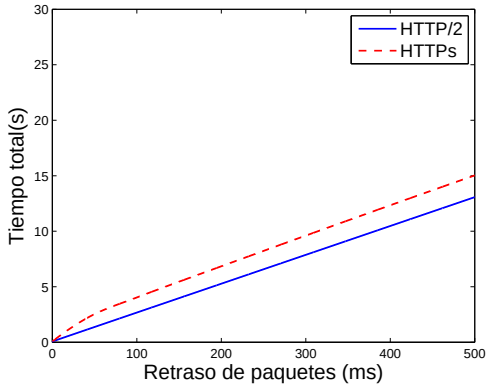
(c) Sesenta y cuatro divisiones

Figura 7.1: Retraso con imagen de 48KB

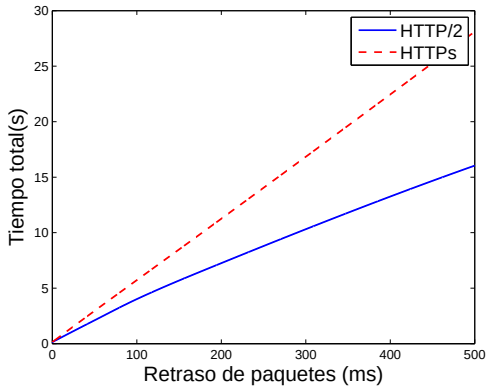
7.1.2.2. Imagen de 578KB



(a) Una división



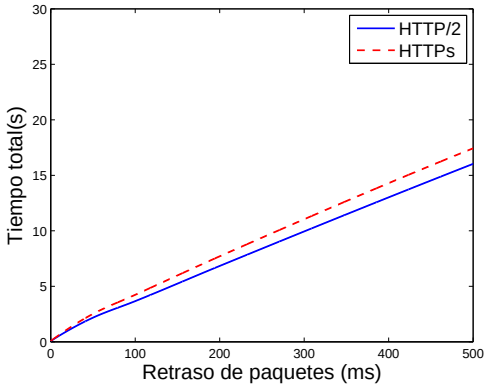
(b) Diez y seis divisiones



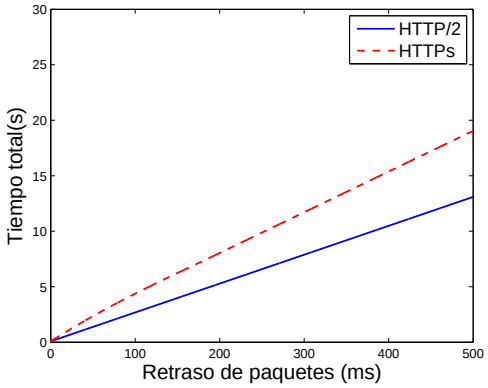
(c) Sesenta y cuatro divisiones

Figura 7.2: Retraso con imagen de 578KB

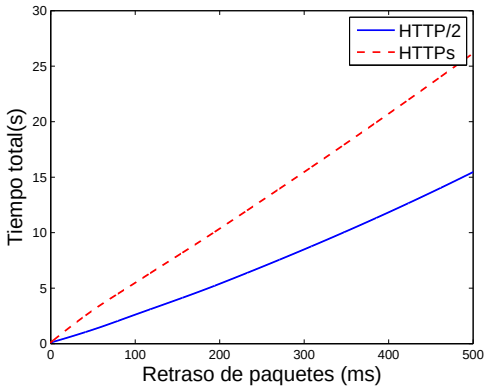
7.1.2.3. Imagen de 1.21MB



(a) Una división



(b) Diez y seis divisiones



(c) Sesenta y cuatro divisiones

Figura 7.3: Retraso con imagen de 1.21MB

7.2. Pérdida de paquetes en la red

7.2.1. Resultados esperados

A priori, HTTP/2 debería verse perjudicado ya que al establecer una sola conexión TCP, si se produjera la pérdida de alguno de los paquetes, el mecanismo de *slow start* afectaría a todos los paquetes enviados.

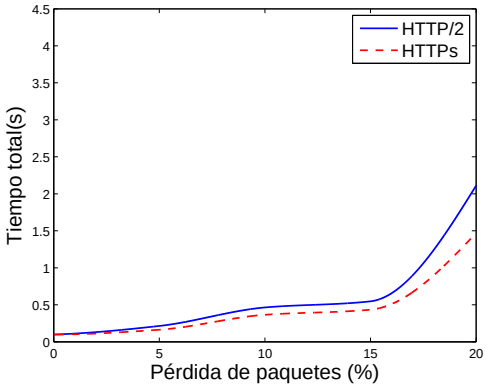
Por otro lado, HTTPS no va a verse tan afectado por la pérdida de paquetes puesto que aunque un recurso se pierda hay más conexiones simultaneas por las que siguen enviándose otros recursos los cuales no se ven afectados. Además, al haber varias conexiones establecidas, hay menos probabilidades de que la pérdida de produzca siempre sobre la misma conexión por lo que los daños provocados por esa pérdida se reparten entre todas las conexiones activas.

7.2.2. Resultados obtenidos

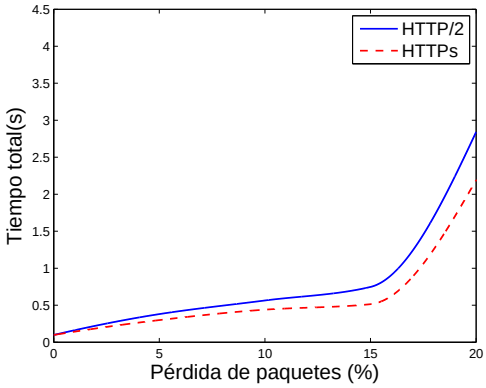
En las figuras 7.4, 7.5 y 7.6 se ha realizado una comparativa del tiempo de carga empleado por HTTP/2 respecto a su versión anterior en función del porcentaje de paquetes perdidos. Para realizar esa comparación, las pruebas se han realizado variando el tamaño de la página web así como el número de recursos de la misma.

Analizando las gráficas, se puede observar que con bajos porcentajes de pérdidas apenas hay diferencias de rendimiento pero que conforme aumentan las pérdidas, aumenta también la diferencia de tiempos entre ambos protocolos. Por otro lado, en las gráficas se observa que cuanto mayor es el tamaño de los datos transmitidos, mayor es la diferencia de rendimiento entre ambos protocolos. Esto se debe a que en HTTP/2 una pérdida afecta a todos los paquetes transmitidos y, cuanto mayores son estos paquetes, más perjudica la acción del mecanismo de *slow start*. Además, se puede comprobar que cuando aumenta el número de recursos empeora el rendimiento de HTTP/2. Por último, al igual que ocurría con las pruebas de los retrasos en la red, sucede algo diferente en la web de más tamaño y es que, como la imagen transmitida es muy grande y supera el tamaño de ventana de TCP, se transmite fragmentada. Por lo tanto,al haber más paquetes viajando por la red, también se produce la pérdida de más paquetes y eso da como resultado que HTTP/2 es menos eficiente que HTTPS. Por lo tanto, estos resultados confirman la hipótesis planteada antes de la elaboración de las pruebas de rendimiento.

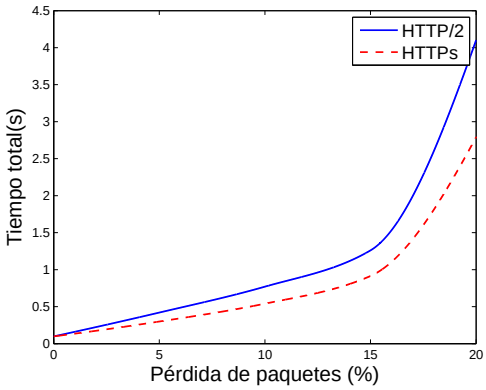
7.2.2.1. Imagen de 48KB



(a) Una división



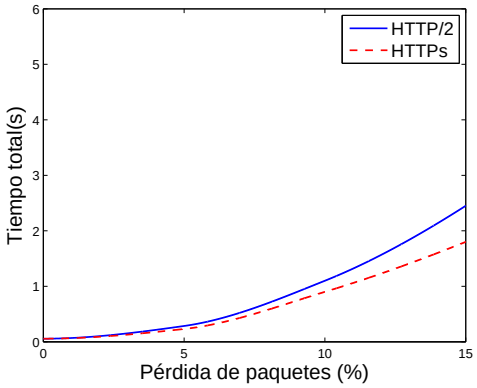
(b) Diez y seis divisiones



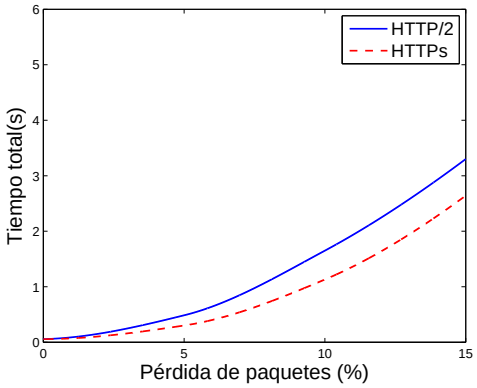
(c) Sesenta y cuatro divisiones

Figura 7.4: Pérdidas con imagen de 48KB

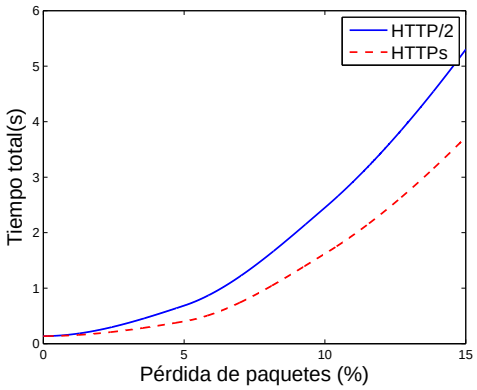
7.2.2.2. Imagen de 578KB



(a) Una división



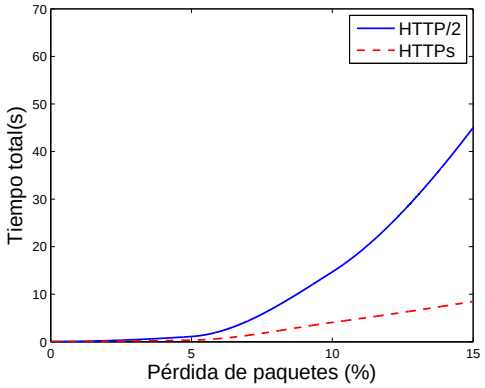
(b) Diez y seis divisiones



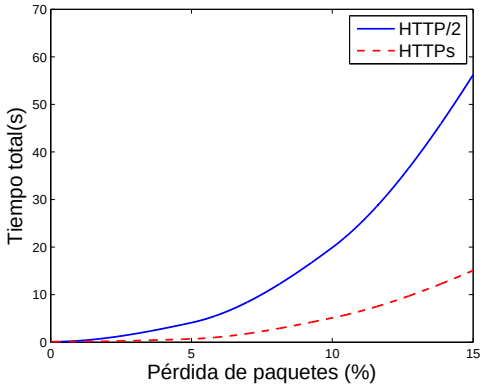
(c) Sesenta y cuatro divisiones

Figura 7.5: Pérdidas con imagen de 578KB

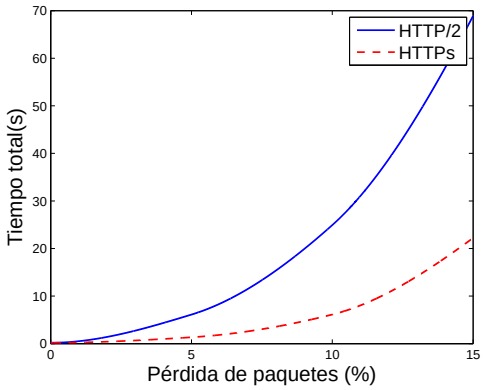
7.2.2.3. Imagen de 1.21MB



(a) Una división



(b) Diez y seis divisiones



(c) Sesenta y cuatro divisiones

Figura 7.6: Pérdidas con imagen de 1.21MB

7.3. Variaciones de ancho de banda en la red

7.3.1. Resultados esperados

Es previsible que en situaciones en las que haya un ancho de banda muy pequeño HTTP/2 mejore el rendimiento de HTTPS debido a que, gracias a HPACK, las cabeceras reducen su tamaño. Por lo tanto, los paquetes van a ser más pequeños permitiendo así que con menores anchos de banda HTTP/2 se vea beneficiado. Además, cuantos más paquetes haya que recibir, mayor será la mejora percibida debido a que habrá que enviar más cabeceras y, por tanto, la reducción del tamaño de los datos recibidos será mayor. En redes con altos anchos de banda no se prevé una gran mejora debido a que se pueden transmitir más datos por la red sin que se vea penalizado el tiempo de carga de las páginas.

7.3.2. Resultados obtenidos

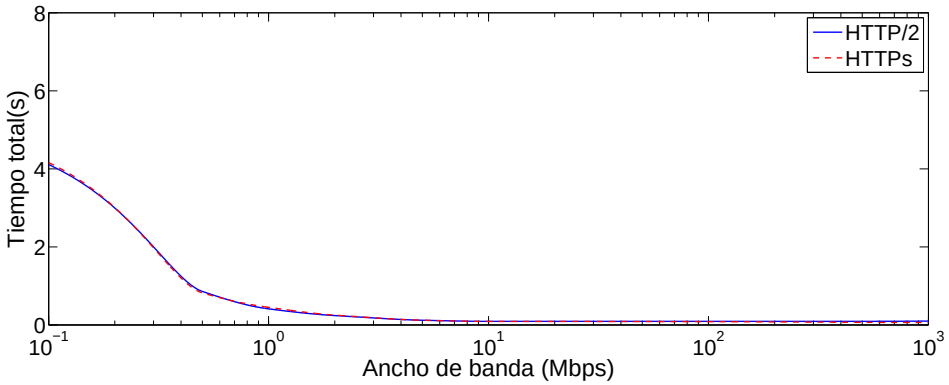
Tal y como se había estimado anteriormente, con anchos de banda pequeños HTTP/2 mejora su rendimiento. Sin embargo, a medida que el ancho de banda aumenta, esa mejora respecto a HTTPS no se percibe con tanta claridad debido a que a mayor ancho de banda, mayor puede ser el tamaño de los datos enviados sin perjudicar al tiempo de carga.

Por otro lado, queda demostrado que a mayor número de recursos, más se aprecia la mejora de HTTP/2 respecto a HTTPS en entornos con anchos de banda pequeños, ya que si se intercambian más peticiones y respuestas, también se están transmitiendo más cabeceras. Esto significa que HPACK puede comprimir más cabeceras cuantos más recursos se transmiten.

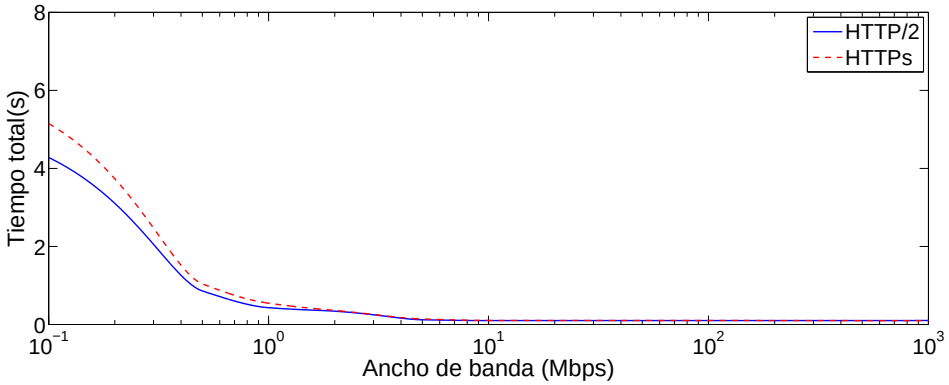
Estos comportamientos pueden observarse en las figuras 7.9, 7.8 y 7.9 las cuales muestran una comparativa sobre como evoluciona el tiempo de carga de las páginas webs a medida que aumenta el ancho de banda.

Además, en la última figura se observa el mismo comportamiento que he citado en las dos pruebas anteriores y es que HTTP/2 mejora a HTTPS incluso cuando la imagen solo está formada por una división. Esto se debe a que la imagen de 1.21 MB es más grande que la ventana TCP y eso hace que el paquete venga fragmentado lo cual hará que se envíen más cabeceras al tener que enviar más paquetes y, por tanto, HPACK podrá comprimir más cabeceras.

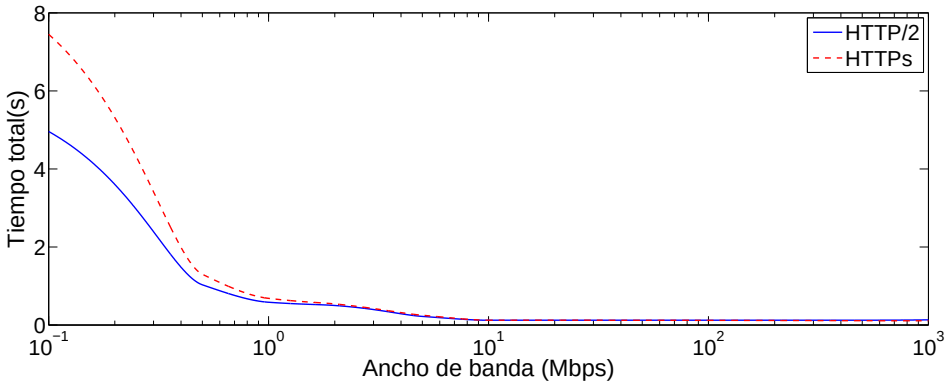
7.3.2.1. Imagen de 48KB



(a) Una división



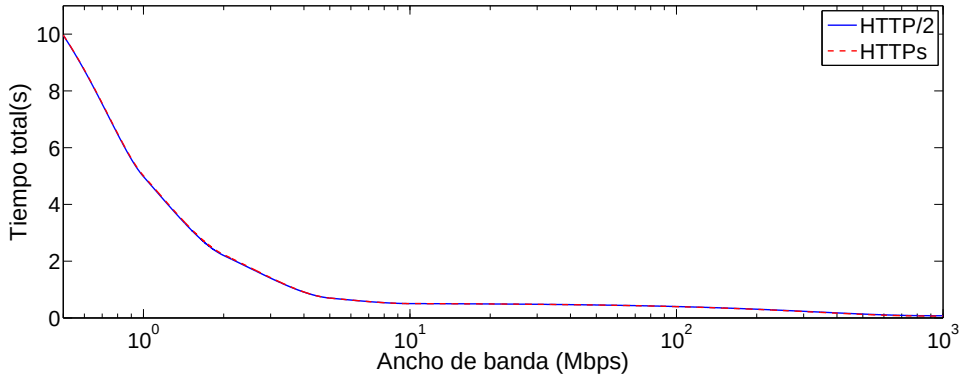
(b) Diez y seis divisiones



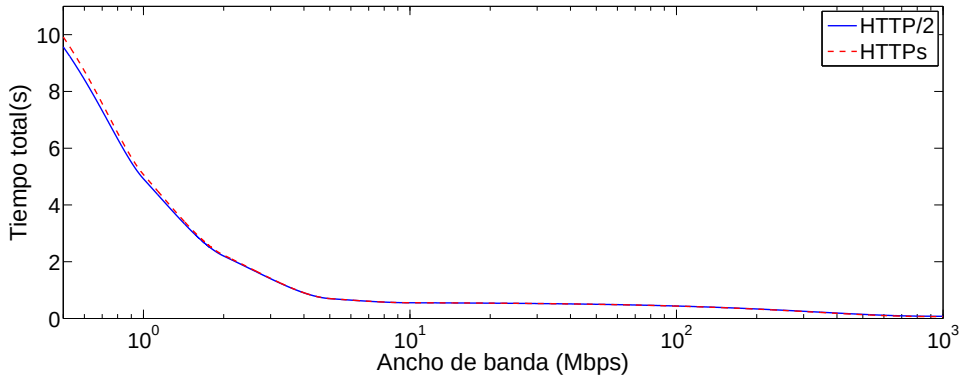
(c) Sesenta y cuatro divisiones

Figura 7.7: Ancho de banda con imagen de 48KB

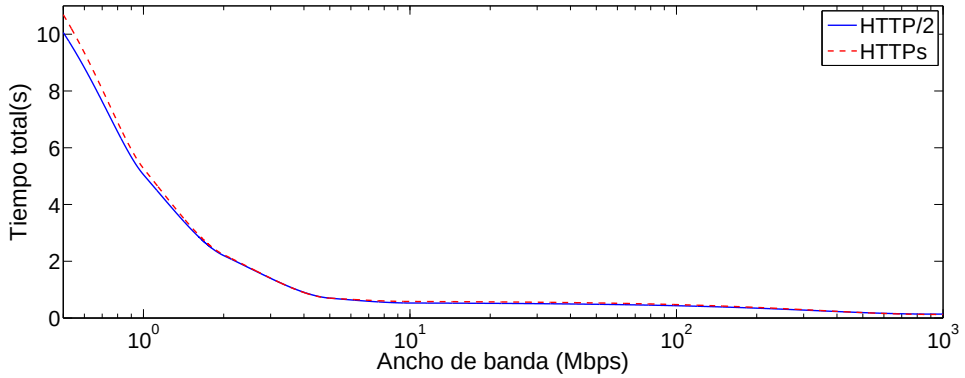
7.3.2.2. Imagen de 578KB



(a) Una división



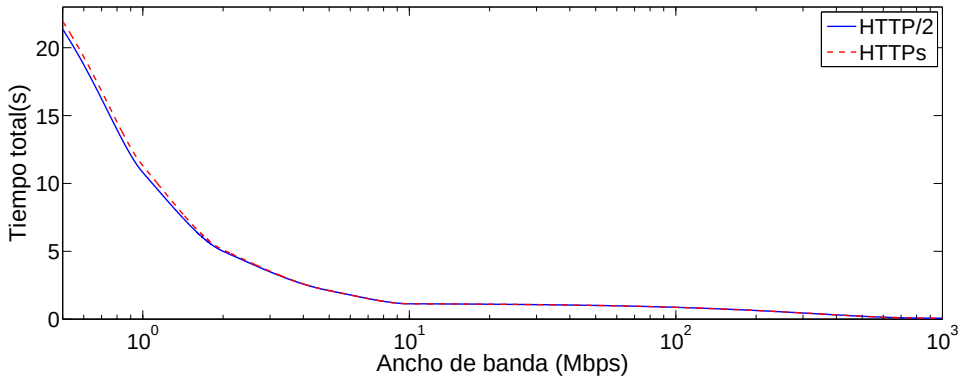
(b) Diez y seis divisiones



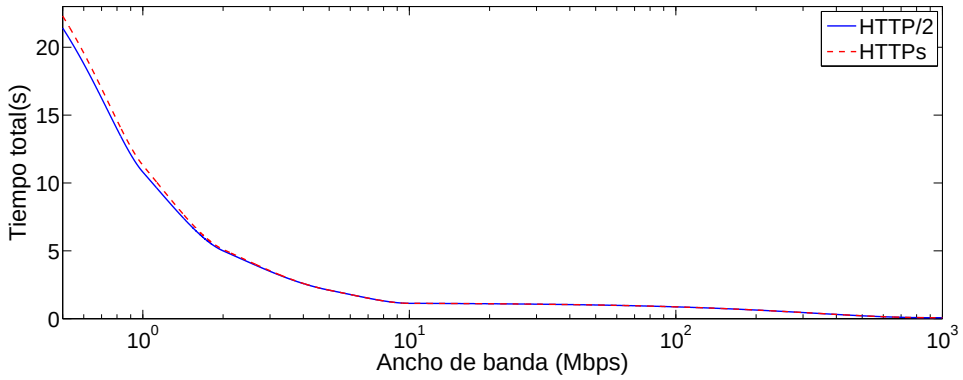
(c) Sesenta y cuatro divisiones

Figura 7.8: Ancho de banda con imagen de 578KB

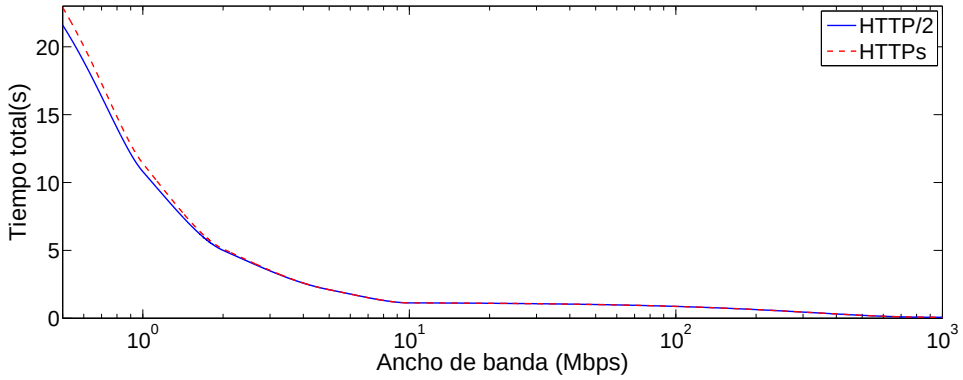
7.3.2.3. Imagen de 1.21MB



(a) Una división



(b) Diez y seis divisiones



(c) Sesenta y cuatro divisiones

Figura 7.9: Ancho de banda con imagen de 1.21MB

7.4. Eficiencia de HPACK

7.4.1. Resultados esperados

Con esta prueba se pretende cuantificar la eficiencia de este protocolo en el servidor Open Lite Speed, ya que la RFC de HPACK no dice como debe realizarse la compresión sino que lo deja a elección de los implementadores. Por lo tanto, este estudio sólo será aplicable al servidor escogido al inicio de este proyecto.

Teniendo en cuenta que este protocolo tiene como objetivo reducir el tamaño de las cabeceras, parece obvio pensar que las pruebas van a demostrar que el tamaño de los paquetes y, por tanto, el tamaño de la web descargada será menor en HTTP/2 que HTTPS en este servidor.

En teoría, cuanto mayor sea el número de peticiones y respuestas, mayor será la compresión ofrecida por HPACK debido a que se estarán enviando más cabeceras duplicadas.

7.4.2. Resultados obtenidos

En los gráficos 7.10, 7.11 y 7.12 se observa una imagen comparativa entre los protocolos HTTP2 y HTTPS en la cual se establece una comparación en el tamaño total descargado en función del número de recursos que forman la página web. Estas imagen muestran que tal y como se había estimado anteriormente, el protocolo HPACK consigue, mediante la compresión de cabeceras, que los datos descargados sean menores en HTTP2.

Por otro lado, mediante las tablas 7.1, 7.2 y 7.3 se establece una comparación del tamaño de cada recurso descargado de forma individual. De esta manera, se puede comprobar que, aunque efectivamente se reduce el tamaño de cada fragmento de imagen recibido, no es una mejora demasiado notable. Esto ocurre porque que el protocolo solo comprime las cabeceras y el tamaño de cada una de ellas no es lo suficientemente grande como para que el ahorro en el tamaño de los recursos recibidos sea demasiado grande.

7.4.2.1. Imagen de 48 KB

Cuadro 7.1: Tamaño de paquetes 48KB

Número de trozos	Tamaño recursos HTTP/2	Tamaño recursos HTTPS
1	48.2KB	48.3KB
16	3.1KB	3.3KB
64	879Bytes	1.1KB

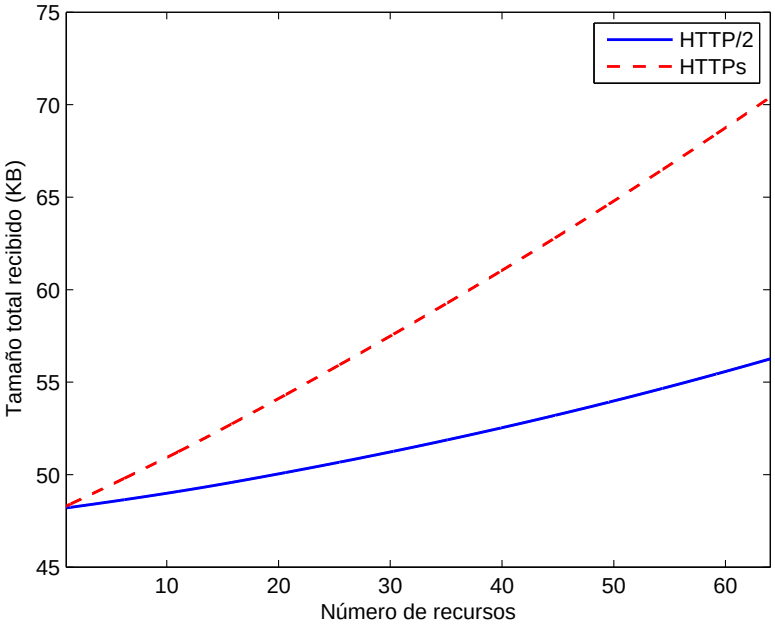


Figura 7.10: HPACK Imagen de 48KB

7.4.2.2. Imagen de 578 KB

Cuadro 7.2: Tamaño de paquetes 578KB

Número de trozos	Tamaño recursos HTTP/2	Tamaño recursos HTTPS
1	579KB	579KB
16	36.3KB	36.4KB
64	9.2KB	9.35KB

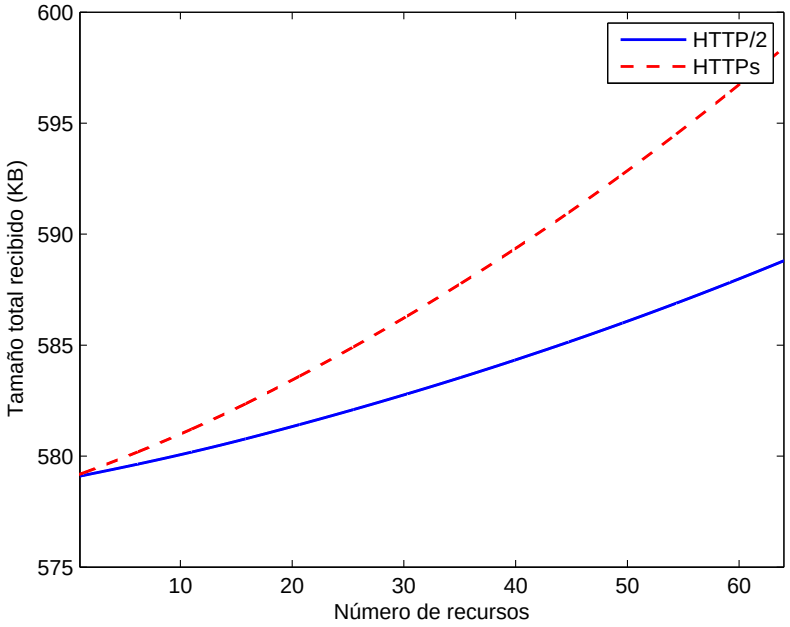


Figura 7.11: HPACK Imagen de 578KB

7.4.2.3. Imagen de 1.21 MB

Cuadro 7.3: Tamaño de paquetes 1.21MB

Número de trozos	Tamaño recursos HTTP/2	Tamaño recursos HTTPS
1	1.2MB	1.21MB
16	75.4KB	76.0KB
64	19.3KB	19.7KB

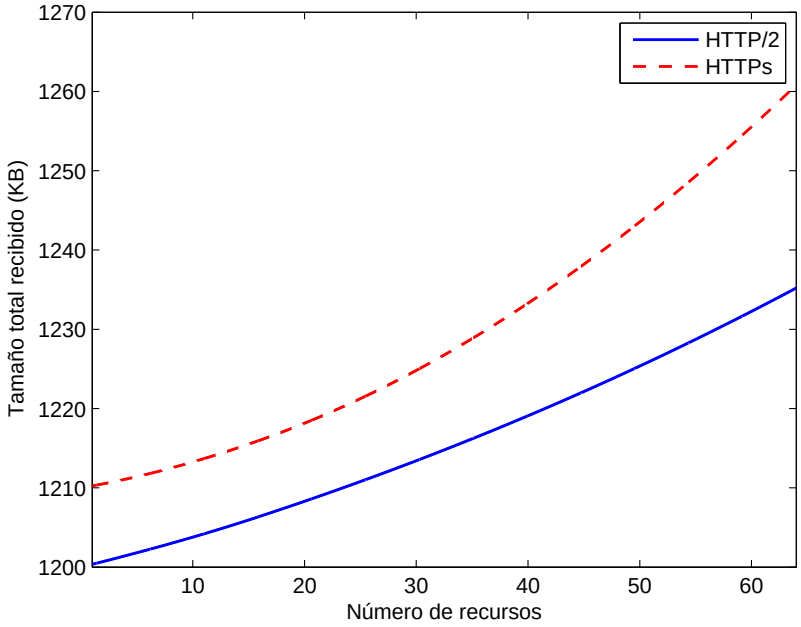


Figura 7.12: HPACK Imagen de 1.21MB

Capítulo 8

Conclusions

8.1. Achievement of objectives

At the beginning of this project I defined the objectives to be met in order to carry out with the main objective which was to evaluate the efficiency of HTTP/2 compared to HTTPS(HTTP/1.1 using TLS). The study of the protocol was not easy because at the start of the project there were neither previous studies of HTTP/2 nor servers nor clients that implemented the protocol.

In spite of this, all of the objectives set have been met.

8.2. Final conclusions

Once I finished the tests and the analysis emulating different network conditions and different webpages my final conclusions are:

- If we try to navigate a webpage with limited resources, HTTP/2 doesn't represent an improvement over the previous version of the protocol.
- If webpages are composed of many resources, HTTP/2 has a better performance than HTTP/1.1.
- In case of suffering loss of packets, HTTP/2 have a worse performance than HTTP/1.1. Because of this, it's possible to guess that in wireless networks, in wich more packets are lost, this protocol will not improve HTTP/1.1 efficiency.
- If packets were suffering delays, HTTP/2 has a better performance than HTTP/1.1.

- The size of the packets decreases due to the header compression introduced by HPACK.
- The loading time is reduced in comparison with HTTPs in environments with low bandwidth. This fact benefit wireless networks wich usually have lower bandwidth than physical networks.

In conclusion, although HTTP/2 improve in some cases the previous version, I don't think that this is such a big improvement as occured before with HTTP/1.1 and HTTP/1.0. Indeed, if the packets are transmitted by wireless technology, HTTP/2 performance may be adversely affected because today's wireless networks support high data rates and, as a consequence, the improvement achieved in low bandwidth disappear and would probably not compensate the poor performance of HTTP/2 when packets are lost.

From my point of view, both protocols should be used because they complement each other very well, HTTP/2 works better in environments without losses and in webpages with a large number of resources, whereas HTTP/1.1 performs better in environments with losses.

8.3. Future works

Different proposals and future improvements based on this project will be listed in the following lines:

- Repeat the performance tests in a wireless environment using WIFI, 3G and 4G networks.
- Make a performance analysis of the push functionality.
- Repeat the performance test using the new QUIC protocol [4] as transport protocol.
- Test the performance of HTTP/2 in different servers like Apache because there were a lot of famous servers that didn't implement HTTP/2 at the start of this project.
- Analyze the performance of HTTP/2 using webpages with different kind of resources such as videos or flash animations and different web technologies like PHP or Java Servlets.

Capítulo 9

Conclusiones

9.1. Consecución de objetivos

Cuando se inició este proyecto de investigación se definieron unos objetivos que debían cumplirse para que se pudiera dar por completado el objetivo principal que era evaluar la eficiencia de HTTP/2 respecto a su versión anterior. El estudio del protocolo fue algo complicado debido a los escasos estudios que existían al inicio de este proyecto, así como la escasez tanto de servidores como de clientes que implementaran este protocolo.

A pesar de ello, se ha conseguido cumplir la totalidad de los objetivos marcados previamente.

9.2. Conclusiones finales

Después de todos los estudios realizados emulando diferentes situaciones de red y diferentes páginas webs, se ha concluido lo siguiente:

- Si se pretende acceder a una web con muy pocos recursos, HTTP/2 no presenta ninguna mejora sobre HTTP/1.1.
- Si una web está formada por muchos recursos, HTTP/2 tiene mejor rendimiento que HTTP/1.1.
- En caso de sufrir pérdida de paquetes, HTTP/2 sale perdiendo frente a su versión anterior por lo que en redes inalámbricas, en las que hay un mayor ratio de pérdidas de paquetes, este protocolo no mejoraría a HTTPs.

- Mejora la velocidad de carga de las páginas web en caso de sufrir retrasos en los paquetes.
- Disminuye el tamaño de los paquetes debido a la compresión de cabeceras itroducida por HPACK.
- La velocidad de carga se ve reducida respecto a HTTPs en entornos con un ancho de banda pequeño. Este hecho beneficia a las redes inalámbricas las cuales suelen disponer de un ancho de banda menor que una red física.

En definitiva, aunque el protocolo mejora en algunos casos a su versión anterior, no considero que realmente sea un cambio tan notable como si lo fue el paso de HTTP/1.0 a HTTP/1.1. De hecho, en en el caso de transmitir los paquetes por redes inalámbricas, HTTP/2 podría salir perdiendo respecto a HTTP/1.1 puesto que como los medios inalámbricos soportan velocidades de transmisión cada vez más elevadas, la mejora que se obtiene al navegar con anchos de banda pequeños desaparecería y no se compensaría el bajo rendimiento que se obtiene al estar en un entorno más propicio a las pérdidas de paquetes. En mi opinión lo ideal sería una convivencia de ambos protocolos puesto que HTTP/2 funciona mejor en entornos sin pérdidas y en páginas con gran número de recursos y HTTP/1.1 funciona mejor en entornos con pérdidas.

9.3. Trabajos futuros

A continuación se van a enumerar distintas propuestas para mejorar y profundizar más en la investigación realizada en este proyecto:

- Realización de las pruebas de rendimiento en un entorno inalámbrico tanto con redes WIFI como 3G y 4G.
- Análisis de rendimiento de la funcionalidad push.
- Evaluación del rendimiento del protocolo HTTP/2 utilizando QUIC [4] como protocolo de transporte en lugar de utilizar TCP.
- Análisis de rendimiento del protocolo HTTP/2 con otros servidores que implementaron la funcionalidad con posterioridad al inicio de esta investigación como Apache.
- Evaluar el rendimiento del protocolo en páginas web con recursos de distintos tipos como vídeos o animaciones flash y diferentes tecnologías web tales como PHP o Java Servlets.

Referencias

[1] HTTP archive. http://httparchive.org/interesting.php. Fecha de último acceso: 05-06-2016.

[2] The internet engineering task force. https://www.ietf.org/. Fecha de último acceso: 05-06-2016.

[3] Ley 32/2003, de 3 de noviembre, general de telecomunicaciones. http://noticias.juridicas.com/base_datos/Admin/l32-2003.t3.html. Fecha de último acceso: 05-06-2016.

[4] QUIC, a multiplexed stream transport over UDP. https://www.chromium.org/quic. Fecha de último acceso: 05-06-2016.

[5] Akamai. BREACH ATTACK. http://https://www.akamai.com/us/en/resources/breach-attack.jsp. Fecha de último acceso: 05-06-2016.

[6] A. Cardaci. Chrome har capturer. https://github.com/cyrus-and/chromehar-capturer. Fecha de último acceso: 05-06-2016.

[7] H. de Saxcé, I. Oprescu, and Y. Chen. Is HTTP/2 really faster than HTTP/1.1? *IEEE Conference on Computer Communications Workshops*, pages 293–299, 2015.

[8] P. Deutsch. DEFLATE compressed data format specification version 1.3. RFC 1951, RFC Editor, Mayo 1996.

[9] Linux Network Developers. Netem. http://www.linuxfoundation.org/collaborate/workgroups/networking/netem. Fecha de último acceso: 05-06-2016.

[10] Linux Network Developers. TC. http://linux.die.net/man/8/tc. Fecha de último acceso: 05-06-2016.

[11] D. Eastlake. Transport layer security (TLS) extensions: Extension definitions. RFC 60661, RFC Editor, Enero 2011.

[12] David A. Huffman. A method for the construction of minimum-redundancy codes. *Proceedings of the IRE*, 40(9):1098–1101, 1952.

[13] R. Peon M. Belshe. Spdy protocol. Technical report, Network Working Group, 2010.

[14] R. Peon M. Belshe. Spdy. Draft draft-mbelshe-httpbis-spdy-00, The Chromium Projects, Febrero 2012.

[15] R. Peon M. Belshe and M. Thomson. Hypertext transfer protocol version 2 (http/2). RFC 7540, RFC Editor, Mayo 2015.

[16] Jitu Padhye and Henrik Frystyk Nielsen. A comparison of SPDY and HTTP performance. *http://research.microsoft.com/apps/pubs/default.aspx?id=170059*, (MSR-TR-2012-102), 2012.

[17] R. Peon and H. Ruellan. Hpack: Header compression for http/2. RFC 7541, RFC Editor, Mayo 2015.

[18] T. Ritter. Details on the CRIME attack. `https://www.nccgroup.trust/us/about-us/newsroom-and-events/blog/2012/september/details-on-the-crime-attack/`, Septiembre 2012. Fecha de último acceso: 05-06-2016.

[19] A. Langley S. Friedl, A. Popov and E. Stephan. Transport layer security (tls) application-layer protocol negotiation extension. RFC 7301, RFC Editor, Julio 2014.

[20] J. Tan and J. Nahata. *PETAL: Preset Encoding Table Information Leakage*. Carnegie Mellon University, Pittsburgh, Pensilvania, 2013.

[21] S. Wei, V. Swaminathan, and Mengbai Xiao. Power efficient mobile video streaming using HTTP/2 server push. *IEEE 17th International Workshop*, pages 1–6, 2015.

[22] G. Tyson Y. Elkhatib and M. Welzl. Can SPDY really make the web faster? *Proceedings of IFIP Networking 2014*, 2014.

Apéndice A

Planificación y presupuesto

A.1. Planificación

A.1.1. Descomposición de tareas

En esta sección se va a desglosar la descomposición de tareas que se han llevado a cabo a la hora de realizar la planificación de este Trabajo de Fin de Grado.

- **Tarea A:** Estudio de los protocolos
  - **Subtarea A.1:** Estudio de HTTP/2.
    - **Descripción:** Estudio de las nuevas características del protocolo
      HTTP/2.
  - **Subtarea A.2:** Estudio de HPACK.
    - **Descripción:** Estudio del protocolo HPACK en combinación con
      HTTP/2.
- **Tarea B:** Despliegue de la red.
  - **Subtarea B.1:** Estudio de los programas de simulación de red.
    - **Descripción:** Estudio de netem y tc como simuladores de red.
  - **Subtarea B.2:** Diseño de la red.
    - **Descripción:** Diseño del routing necesario para utilizar un nodo intermedio como simulador de red.

- **Subtarea B.3:** Despliegue físico de las máquinas.
  - **Descripción:** Despliegue de tres nodos en el despacho. Dos se utilizarán como cliente-servidor y un tercer nodo que hará función de intermediario para simular la red.
- **Subtarea B.4:** Configuración de la red.
  - **Descripción:** Aplicación del routing diseñado en la tarea B.2 y configuración del nodo intermedio como simulador de red con los programas analizados en la tarea B.1

- **Tarea C:** Instalación de software.
  - **Subtarea C.1:** Instalación del servidor HTTP/2.
    - **Descripción:** Estudio e instalación del servidor (OpenLiteSpeed) en la máquina habilitada como servidor.
  - **Subtarea C.2:** Instalación del cliente HTTP/2
    - **Descripción:** Estudio e instalación del navegador elegido para las pruebas (Google Chrome) tanto en el entorno Linux como en el entorno Windows.
  - **Subtarea C 3:** Instalación y configuración de Wireshark.
    - **Descripción:** Instalación de Wireshark en el cliente y configuración
      del mismo para capturar paquetes HTTP/2.

- **Tarea D:** Desarrollo de script para pruebas.
  - **Descripción:** Desarrollo en bash y node.js del script de pruebas automatizadas que permitirá medir el tiempo empleado para descargar cada página.

- **Tarea E:** Obtención y análisis de los resultados.
  - **Subtarea E.1:** Obtención de los resultados en un documento Excel.
    - **Descripción:** Clasificar los resultados obtenidos mediante el script elaborado en la tarea D en documentos Excel.
  - **Subtarea E.2:** Elaboración de gráficos para el análisis de resultados.

        - **Descripción:** Generación de gráficos comparativos utilizando Matlab para el posterior análisis de los resultados.

- Tarea F: Elaboración de la memoria del Trabajo de Fin de Grado.

    - Subtarea F.1: Organización y estructura de la memoria.
        - **Descripción:** Diseño del esquema y estructura que tendrá la memoria.
    - Subtarea F.2: Preparación de las 5 imágenes que se van a utilizar en la memoria.
        - **Descripción:** Elección de las gráficas a añadir en la memoria así como la obtención de las imágenes necesarias para la elaboración de la memoria.
    - Subtarea F.3: Redacción de la memoria.
        - **Descripción:** Elaboración de la memoria.
    - Subtarea F.4: Redacción del resumen en inglés de la memoria.
        - **Descripción:** Redacción del resumen en inglés de la memoria.

A.1.2. Diagrama de Gantt

	Nombre	Duración	Inicio	Fin	Predecesoras	Recursos
1	Estudio de HTTP/2	7d	11/01/2016	19/01/2016		Daniel Vega
2	Estudio de HPACK	7d	20/01/2016	28/01/2016	1	Daniel Vega
3	Estudio de los programas de simulación de red	7d	29/01/2016	08/02/2016	2	Daniel Vega
4	Diseño de la red	5d	09/02/2016	15/02/2016	3	Daniel Vega
5	Despliegue físico de las máquinas	3d	16/02/2016	18/02/2016	4	Subcontrata
6	Configuración de la red	3d	19/02/2016	23/02/2016	5	Daniel Vega
7	Instalación del servidor HTTP/2	6d	24/02/2016	02/03/2016	6	Subcontrata
8	Instalación del cliente HTTP/2	6d	24/02/2016	02/03/2016	6	Daniel Vega
9	Instalación y configuración de wireshark	5d	03/03/2016	09/03/2016	7,8	Daniel Vega
10	Desarrollo de script para pruebas	14d	10/03/2016	29/03/2016	7,8,9	Daniel Vega
11	Obtención de los resultados en un documento Excel	3d	30/03/2016	01/04/2016	10	Daniel Vega
12	Elaboración de gráficos para el análisis de resultados.	3d	04/04/2016	06/04/2016	11	Daniel Vega
13	Organización y estructura de la memoria	3d	07/04/2016	11/04/2016	12	Daniel Vega
14	Preparación de las imágenes que se van a utilizar en la memoria	3d	12/04/2016	14/04/2016	12,13	Daniel Vega
15	Redacción de la memoria	31d	15/04/2016	27/05/2016	14	Daniel Vega
16	Redacción del resumen en inglés de la memoria	7d	30/05/2016	07/06/2016	15	Daniel Vega

Figura A.1: Tareas del diagrama de Gantt

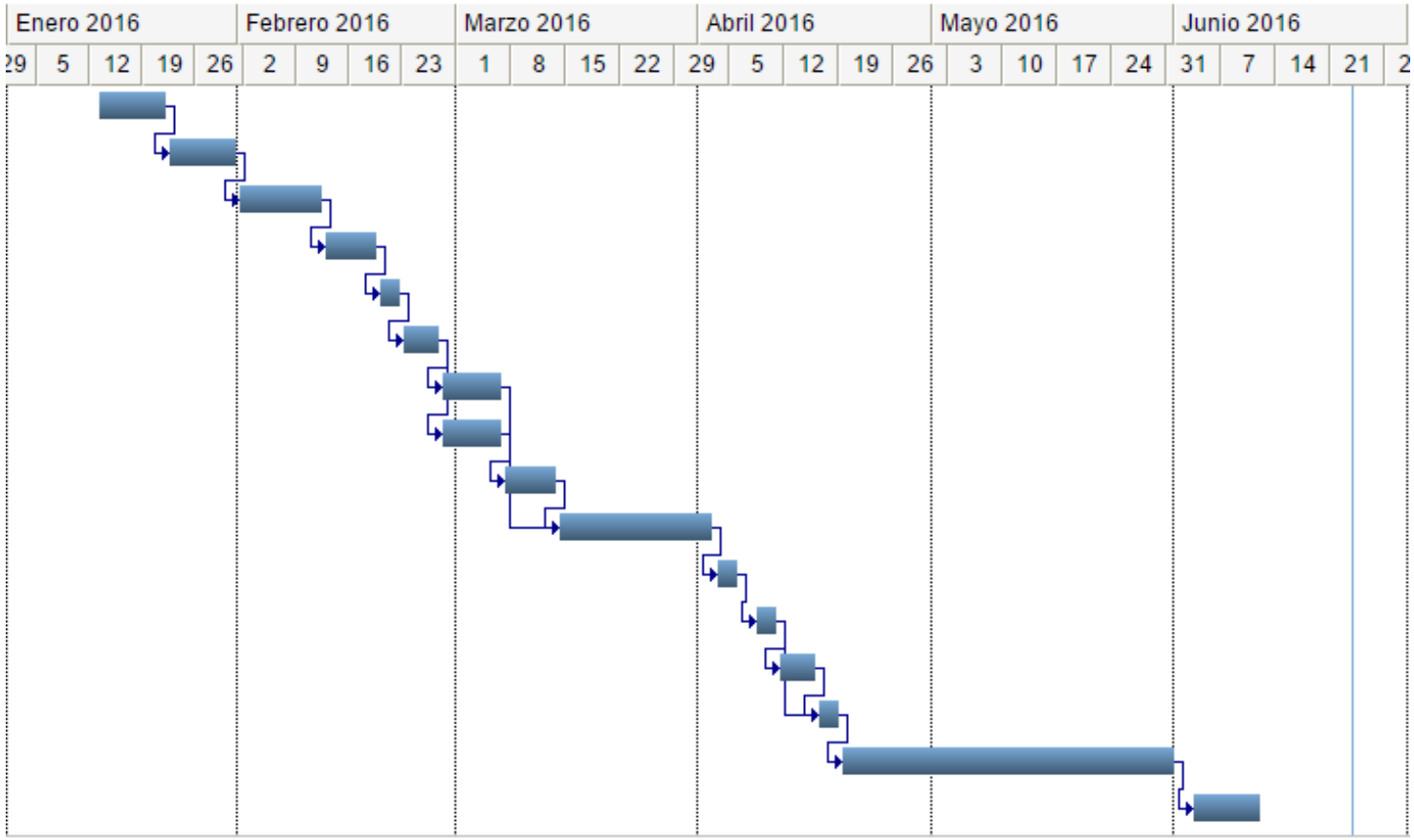


Figura A.2: Diagrama de Gantt

A.2. Presupuesto

1. Autor:
   Daniel Vega Jiménez

2. Departamento:
   Ingeniería Telemática

3. Descripción del proyecto:

Título:	Estudio de prestaciones del protocolo HTTP versión 2
Duración (meses):	6
Tasa de costes indirectos:	20 %

4. Presupuesto total del proyecto (valores en Euros):
   14.283,60 Euros

5. Desglose presupuestario (costes directos):

PERSONAL

Nombre	DNI	Categoría	Dedicación (meses)	Coste (Euros)	Coste imputable
Vega Jiménez, Daniel	47541357N.	Ingeniero Junior	6 meses	1428,57	8571,42
				Total	8571.42

EQUIPOS

Descripción	Coste (Euros)	% uso dedicado al proyecto	Dedicación (meses)	Período depreciación	Coste imputable
Ordenador sobre mesa Windows	550,00	100	6	60	55,00€
Ordenador sobre mesa Linux	550,00	100	6	60	55,00
Servidor	740,00	100	2	60	24,67
Máquina virtual	70,00	100	2	60	2,33
				Total	82,00

SUBCONTRATACIÓN DE TAREAS

Descripción	Empresa	Coste imputable
Mantenimiento del servidor	UC3M	3.000,00
	Total	3.000,00

OTROS COSTES DIRECTOS DEL PROYECTO		
Descripción	Empresa	Coste imputable
Material Fungible	UC3M	50,00
Internet	UC3M	200,00
	Total	250,00

6. Resumen de costes:

Costes directos	
Personal	8571,00
Amortización	82,00
Subcontratación de tareas	3.000,00
Costes de funcionamiento	250,00
Costes indirectos	2.380,60
Total	14.283,60

El presupuesto total de este proyecto asciende a la cantidad de 14.283,60 EUROS.

Leganés a 08 de Junio de 2016

El ingeniero proyectista

Fdo. Daniel Vega Jiménez